CSE 2017 Data Structures and Lab

Lecture #13: Sorting

Eun Man Choi

Sorting means . . .

- The values stored in an array have keys of a type for which the relational operators are defined. (We also assume unique keys.)

- Sorting rearranges the elements into either ascending or descending order within the array. (We'll use ascending order.)

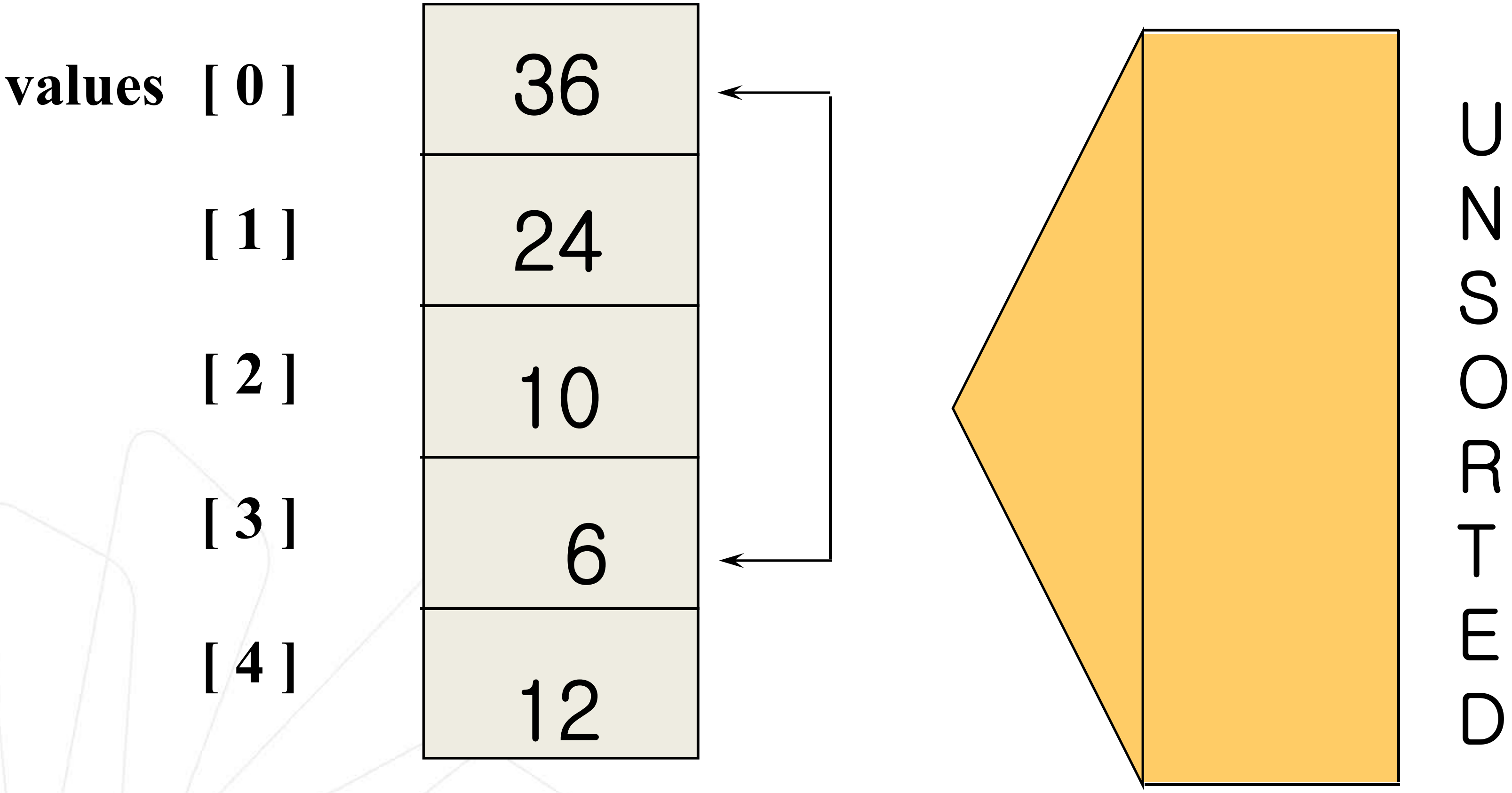
Straight Selection Sort

values	[0]	36
	[1]	24
	[2]	10
	[3]	6
	[4]	12

Divides the array into two parts: already sorted, and not yet sorted.

On each pass, finds the smallest of the unsorted elements, and swaps it into its correct place, thereby increasing the number of sorted elements by one.

Selection Sort: Pass One



values [0] 36

[1] 24

[2] 10

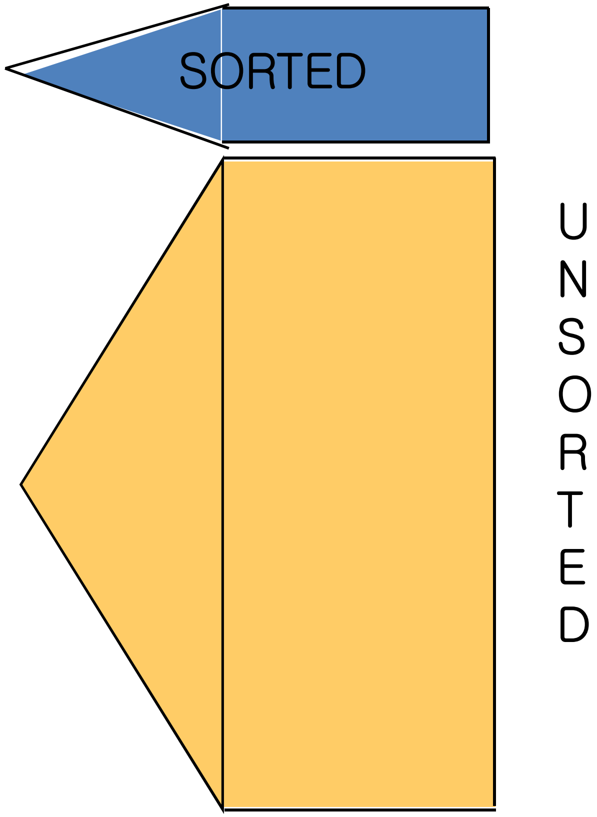
[3] 6

[4] 12

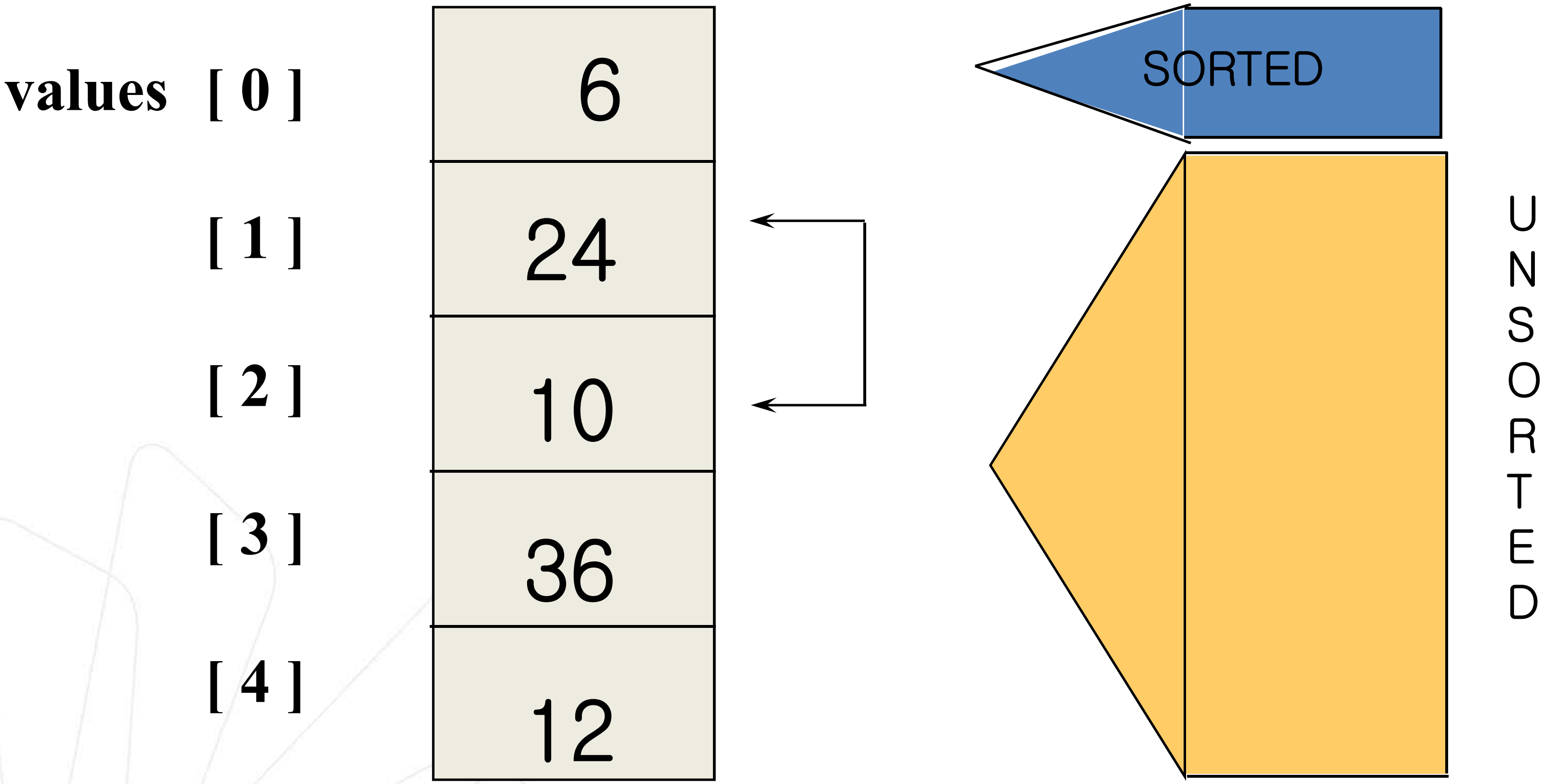
UNSORTED

Selection Sort: End Pass One

values	
[0]	6
[1]	24
[2]	10
[3]	36
[4]	12


SORTED
UNSORTED

Selection Sort: Pass Two

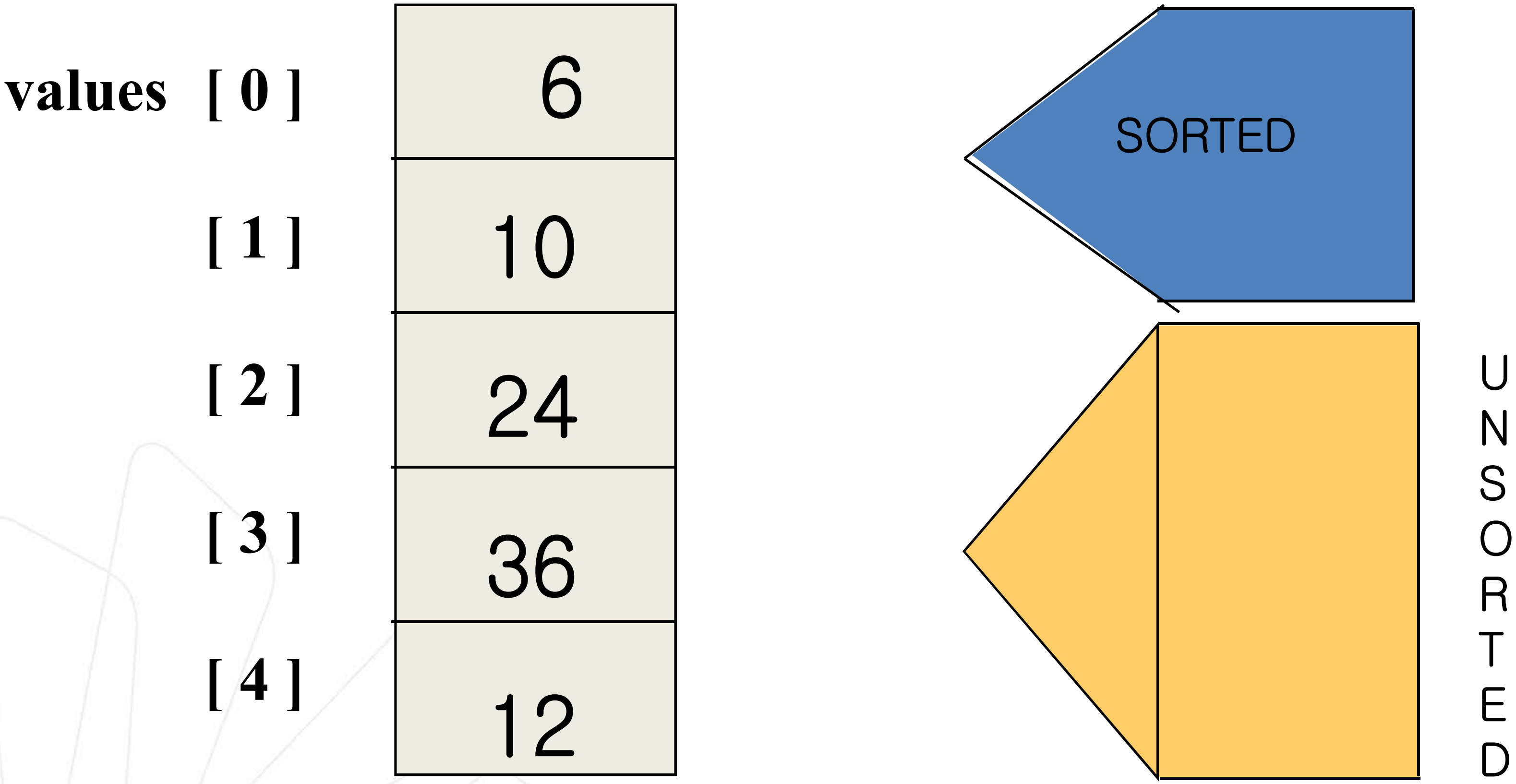


values	
[0]	6
[1]	24
[2]	10
[3]	36
[4]	12

SORTED

UNSORTED

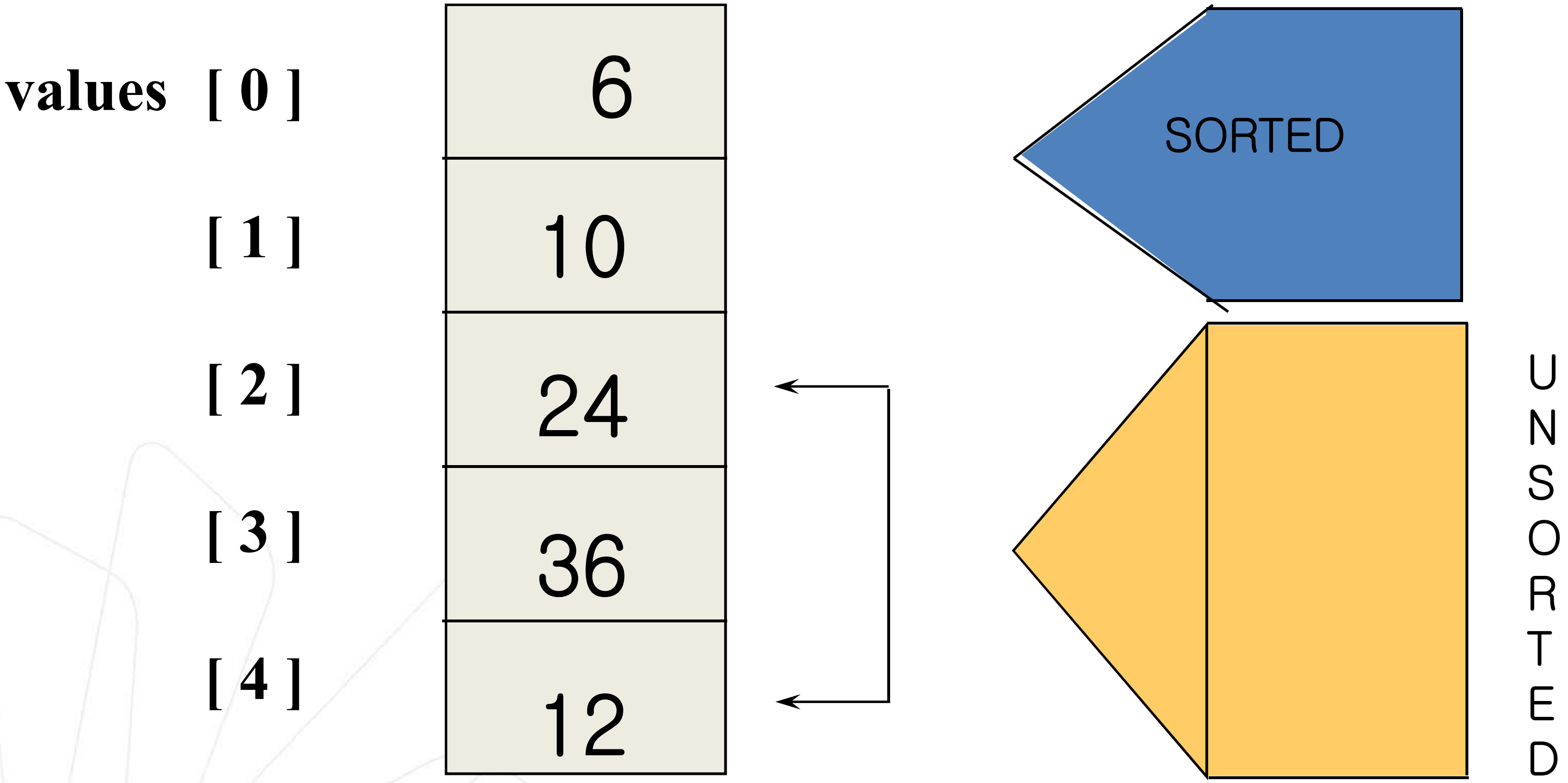
Selection Sort: End Pass Two



values	
[0]	6
[1]	10
[2]	24
[3]	36
[4]	12

SORTED
UNSORTED

Selection Sort: Pass Three



values [0] 6

[1] 10

[2] 24

[3] 36

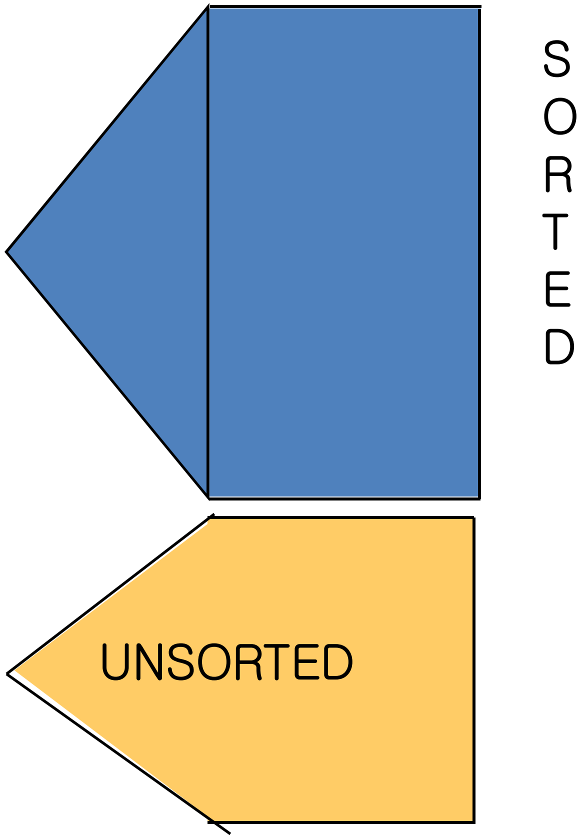
[4] 12

SORTED

UNSORTED

Selection Sort: End Pass Three

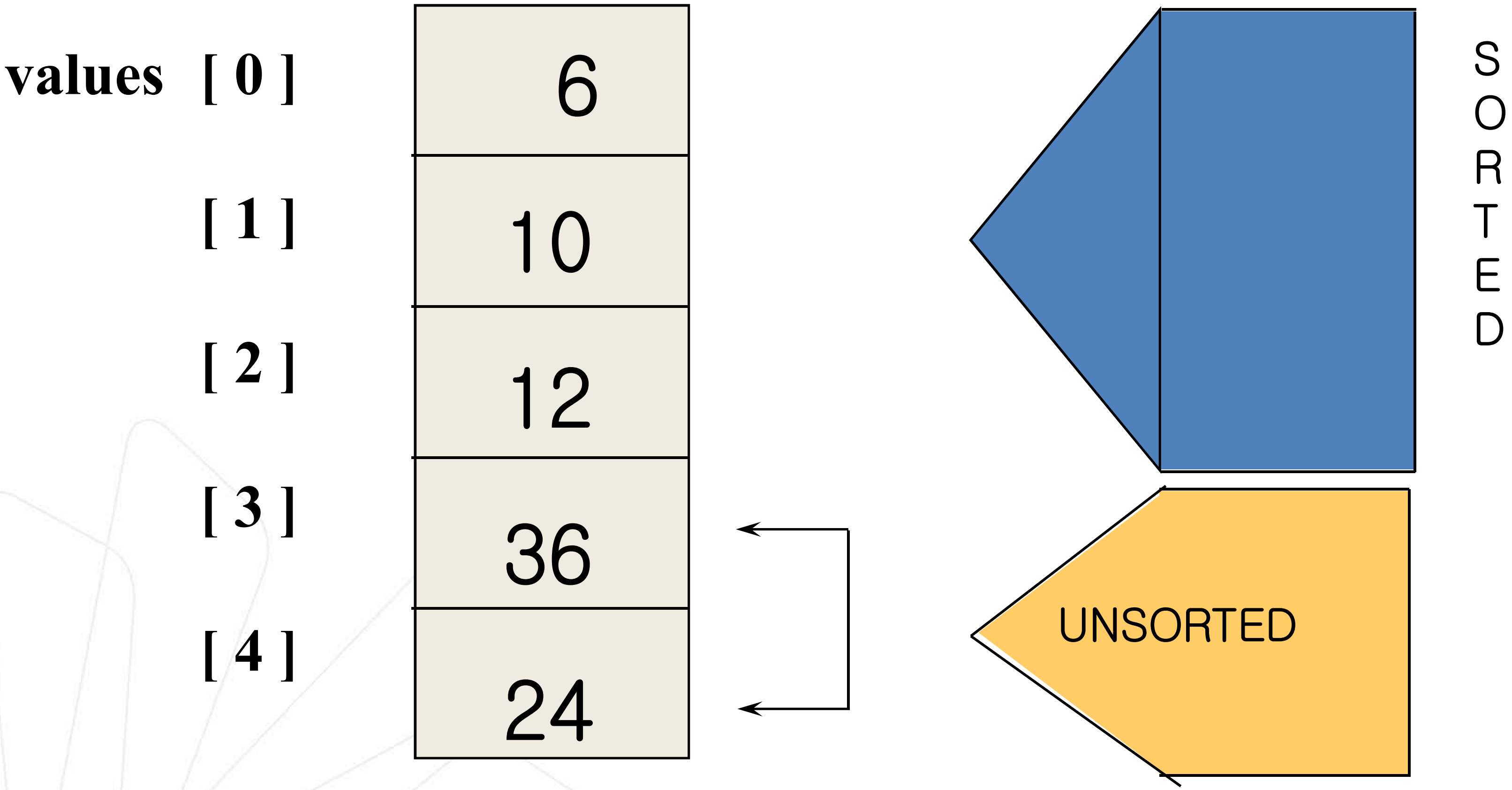
values	[0]	6
	[1]	10
	[2]	12
	[3]	36
	[4]	24



SORTED

UNSORTED

Selection Sort: Pass Four



values [0] 6

[1] 10

[2] 12

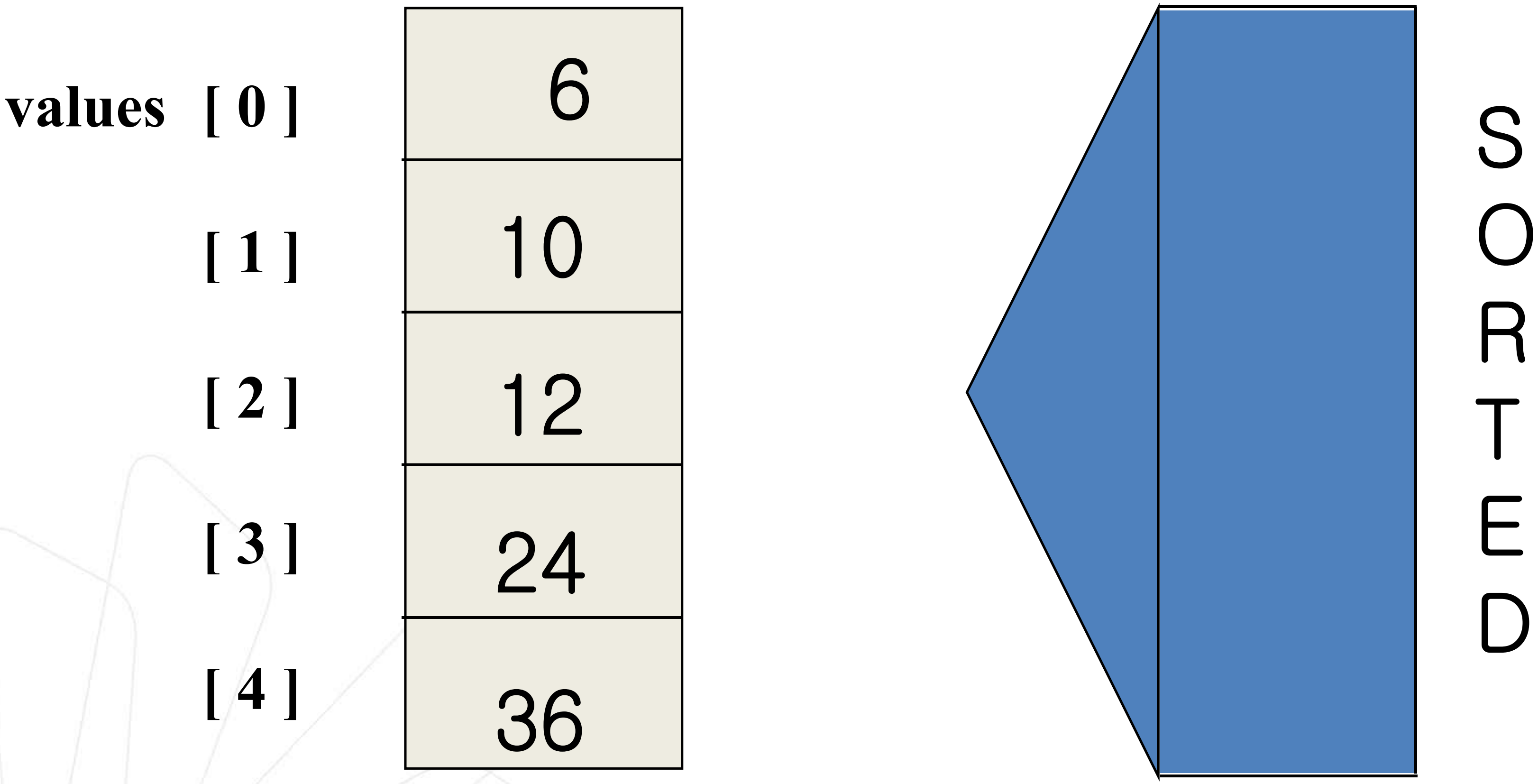
[3] 36

[4] 24

SORTED

UNSORTED

Selection Sort: End Pass Four



values	
[0]	6
[1]	10
[2]	12
[3]	24
[4]	36

Selection Sort: How many comparisons?

values	[0]	6
	[1]	10
	[2]	12
	[3]	24
	[4]	36

4 compares for values[0]

3 compares for values[1]

2 compares for values[2]

1 compare for values[3]

= 4 + 3 + 2 + 1

For selection sort in general

- **The number of comparisons when the array contains N elements is**

$$\text{Sum} = (N-1) + (N-2) + \ldots + 2 + 1$$

Notice that . . .

$$\text{Sum} = (N-1) + (N-2) + . . . + 2 + 1$$

$$+ \ \text{Sum} = 1 + 2 + . . . + (N-2) + (N-1)$$

$$2* \text{Sum} = \underbrace{N + N + . . . + N + N}$$

$$2 * \text{Sum} = N * (N-1)$$

$$\boxed{\text{Sum} = \frac{N * (N-1)}{2}}$$

For selection sort in general

- **The number of comparisons when the array contains N elements is**

$$\text{Sum} = (N-1) + (N-2) + \ldots + 2 + 1$$

$$\text{Sum} = N * (N-1) / 2$$

$$\text{Sum} = .5\,N^2 - .5\,N$$

$$\text{Sum} = O(N^2)$$

```
template <class  ItemType >
int  MinIndex  ( ItemType  values [ ] ,   int  start ,  int end )
//  Post: Function value = index of the smallest value in
//                values [start]  . . values [end].
{
  int  indexOfMin = start ;
  for  ( int  index = start + 1 ;  index <= end ; index++ )
        if  ( values [ index ] < values [ indexOfMin ] )
              indexOfMin = index ;

  return   indexOfMin;
}
```

```
template <class  ItemType >
void  SelectionSort  ( ItemType  values [ ] ,   int  numValues )


//  Post: Sorts array values[0 . . numValues-1 ] into ascending
//              order by key
{
  int  endIndex = numValues - 1 ;

  for  ( int  current = 0 ; current < endIndex ; current++ )

         Swap ( values [ current ] ,
                        values [ MinIndex ( values, current, endIndex ) ]
  ) ;

}
```

Bubble Sort

values	[0]	36
	[1]	24
	[2]	10
	[3]	6
	[4]	12

Compares neighboring pairs of array elements, starting with the last array element, and swaps neighbors whenever they are not in correct order.

On each pass, this causes the smallest element to "bubble up" to its correct place in the array.

```
template <class  ItemType >
void BubbleUp  ( ItemType  values [ ] ,   int  start ,  int end )

//  Post: Neighboring elements that were out of order have been
//             swapped between values [start] and  values [end],
//             beginning at values [end].
{
  for  ( int  index = end ;  index > start ; index-- )
        if  (values [ index ] < values [ index - 1 ] )
              Swap ( values [ index ], values [ index - 1 ] ) ;
}
```

```
template <class  ItemType >
void  BubbleSort  ( ItemType  values [ ] ,   int  numValues )

//  Post: Sorts array values[0 . . numValues-1 ] into ascending
//              order by key
{
  int  current = 0  ;

  while  ( current < numValues - 1 )

          BubbleUp ( values , current , numValues - 1 ) ;

          current++ ;

}
```

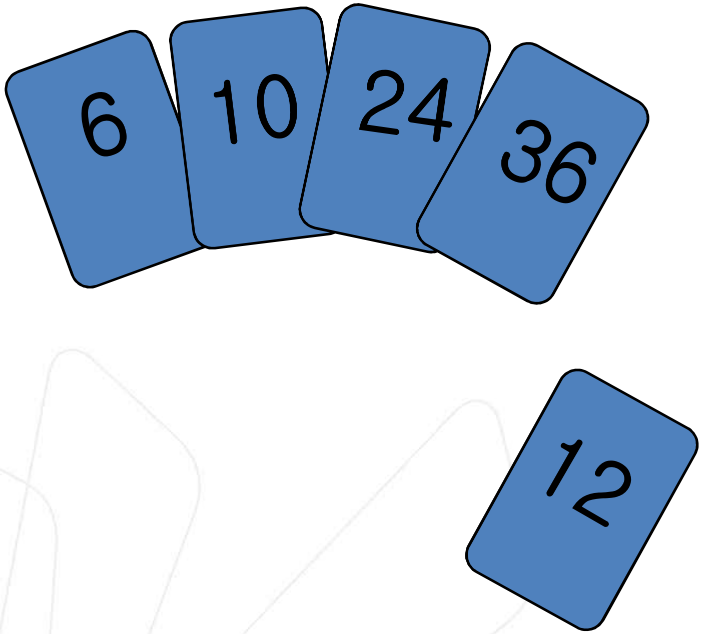
Insertion Sort

values	[0]	36
	[1]	24
	[2]	10
	[3]	6
	[4]	12

One by one, each as yet unsorted array element is inserted into its proper place with respect to the already sorted elements.

On each pass, this causes the number of already sorted elements to increase by one.

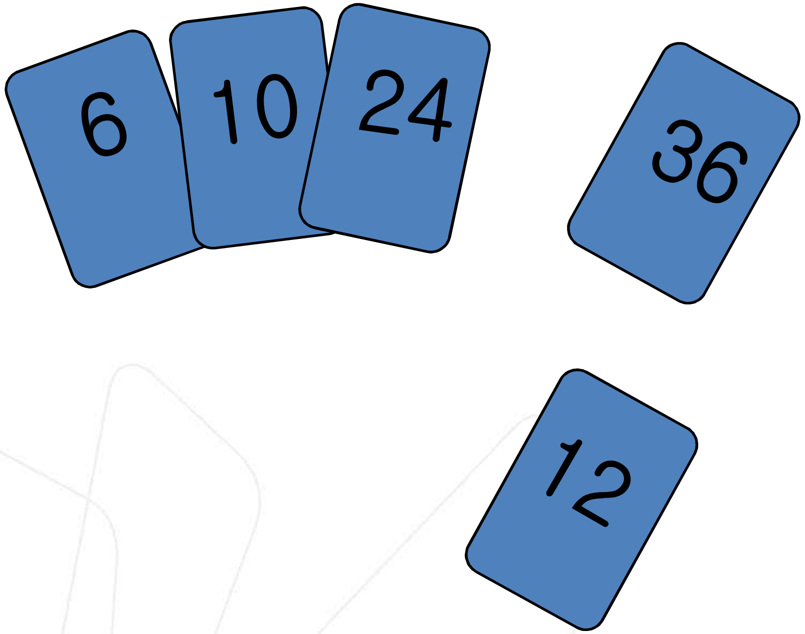
Insertion Sort



Works like someone who "inserts" one more card at a time into a hand of cards that are already sorted.

To insert 12, we need to make room for it by moving first 36 and then 24.

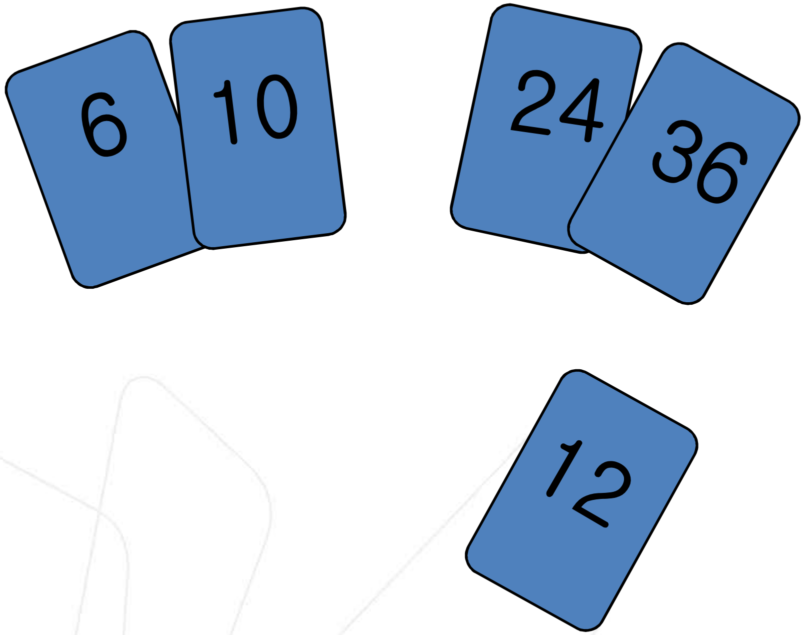
Insertion Sort



Works like someone who “inserts” one more card at a time into a hand of cards that are already sorted.

To insert 12, we need to make room for it by moving first 36 and then 24.

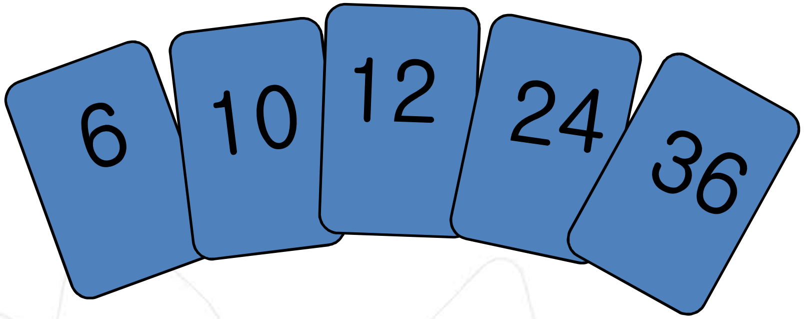
Insertion Sort



Works like someone who "inserts" one more card at a time into a hand of cards that are already sorted.

To insert 12, we need to make room for it by moving first 36 and then 24.

Insertion Sort



Works like someone who "inserts" one more card at a time into a hand of cards that are already sorted.

To insert 12, we need to make room for it by moving first 36 and then 24.

```
template <class  ItemType >
void InsertItem  ( ItemType  values [ ] ,   int  start ,  int end

//  Post: Elements between values [start] and  values [end]
//             have been sorted into ascending order by key.
{
  bool  finished = false ;
  int     current  =  end ;
  bool  moreToSearch = ( current != start ) ;

  while  ( moreToSearch  &&  !finished )
  {

        if  (values [ current ] < values [ current - 1 ] )
        {
              Swap ( values [ current ], values [ current - 1 ] ) ;
              current-- ;
              moreToSearch = ( current != start );
        }
        else
              finished = true ;
  }
}
```

```
template <class  ItemType >
void  InsertionSort  ( ItemType  values [ ] ,   int  numValues )

//  Post: Sorts array values[0 . . numValues-1 ] into ascending
//              order by key
{
  for  ( int count = 0 ; count < numValues ; count++ )

          InsertItem ( values , 0 , count ) ;

}
```

Sorting Algorithms and Average # of Comparisons

Simple Sorts

- Straight Selection Sort
- Bubble Sort
- Insertion Sort

$O(N^2)$

More Complex Sorts

- Quick Sort
- Merge Sort
- Heap Sort

O(N*log N)

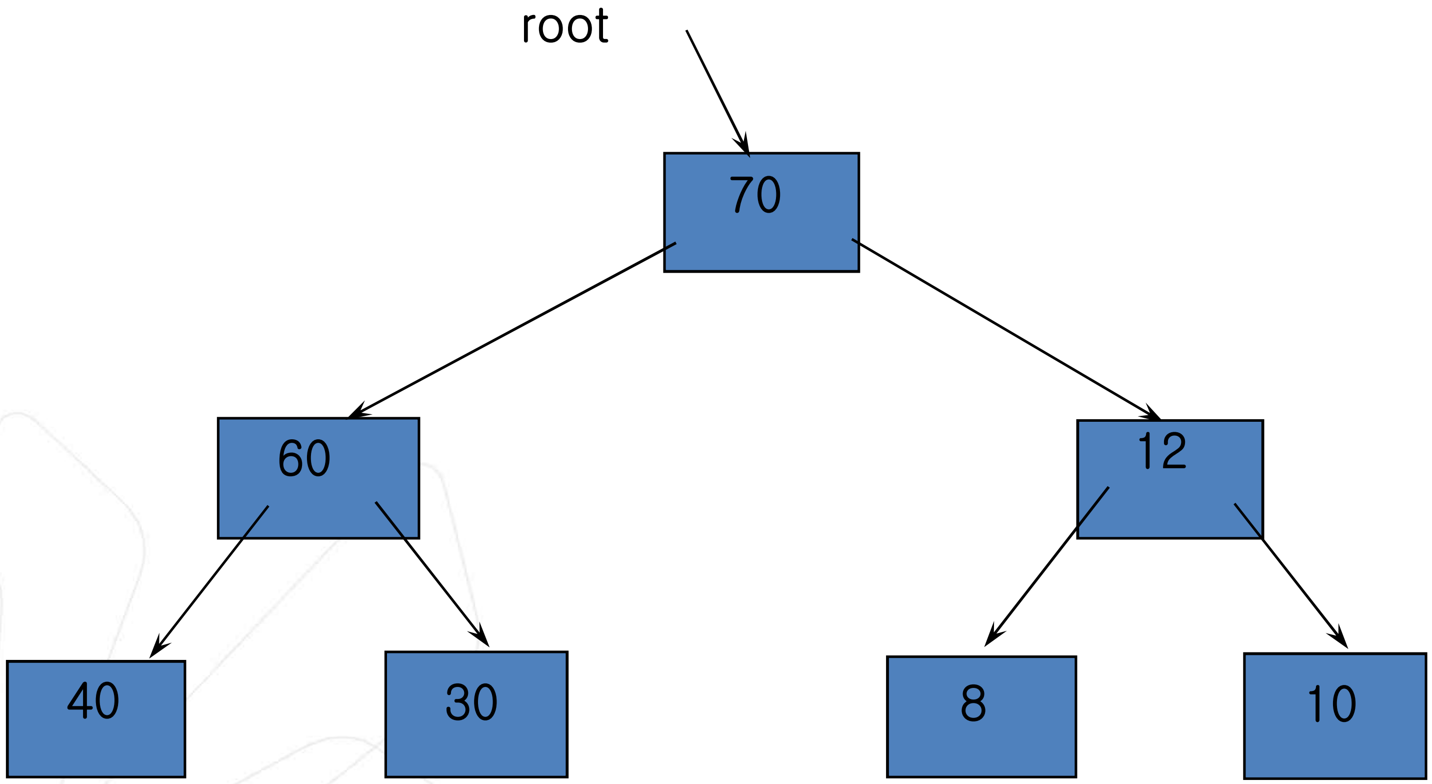
Recall that . . .

A heap is a binary tree that satisfies these special SHAPE and ORDER properties:

- Its shape must be a complete binary tree.
- For each node in the heap, the value stored in that node is greater than or equal to the value in each of its children.

Heap Sort

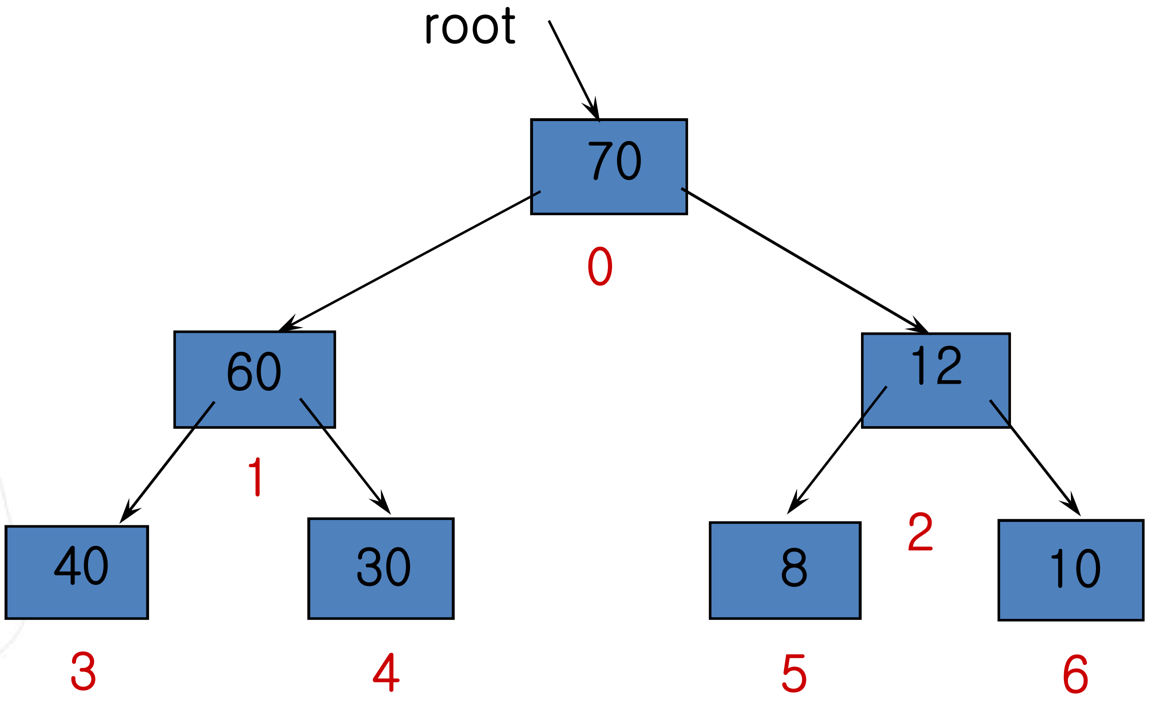
- The largest element in a heap is always found in the root node


root
70
60
12
40
30
8
10

Heap Sort

- **The heap can be stored in an array**

	values
[0]	70
[1]	60
[2]	12
[3]	40
[4]	30
[5]	8
[6]	10

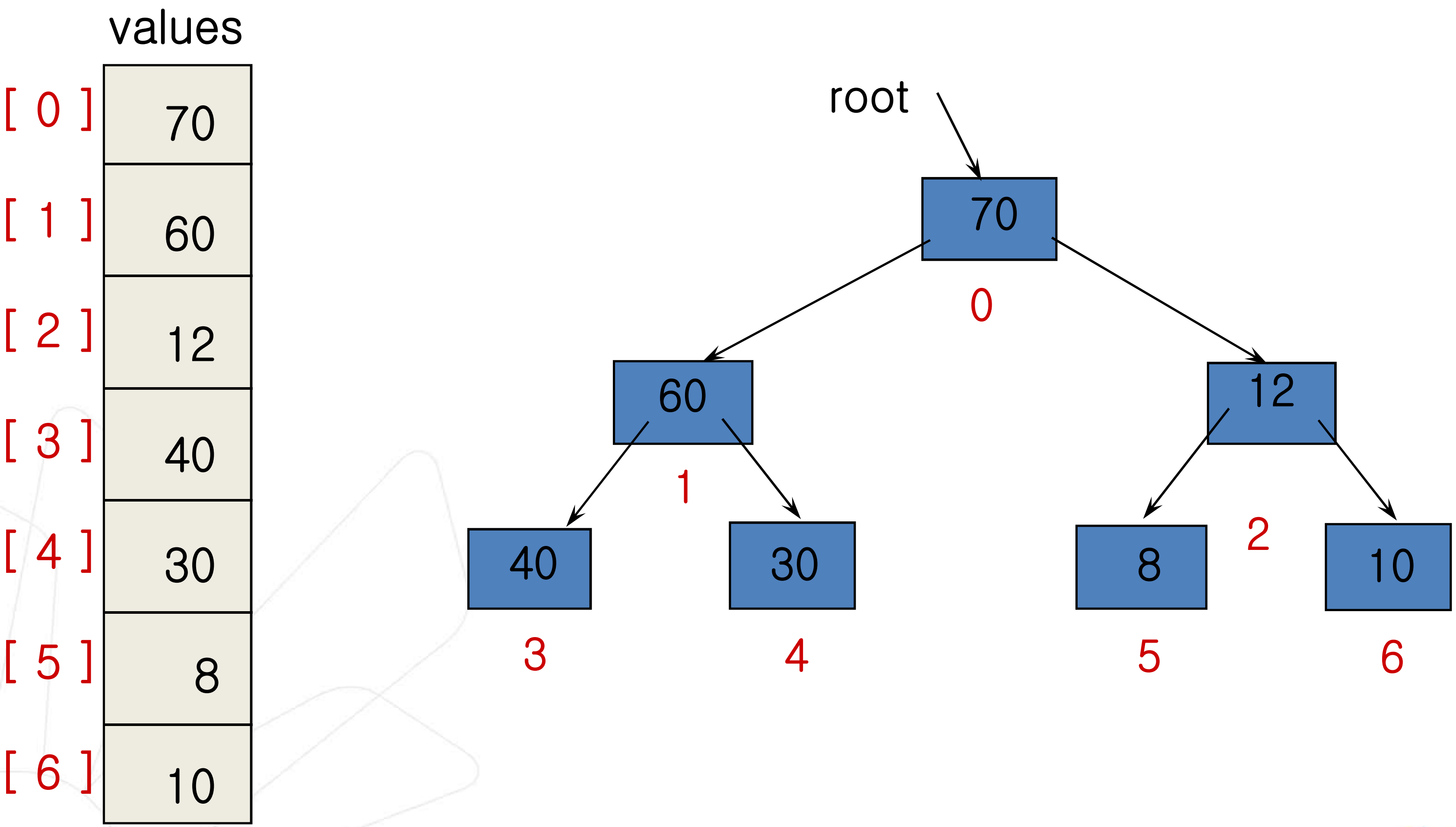

root
70
0
60
1
12
2
40
3
30
4
8
5
10
6

Heap Sort Approach

- First, make the unsorted array into a heap by satisfying the order property. Then repeat the steps below until there are no more unsorted elements.
- Take the root (maximum) element off the heap by swapping it into its correct place in the array at the end of the unsorted elements.
- Reheap the remaining unsorted elements. (This puts the next-largest element into the root position).

Heap Sort

- **After creating the original heap**

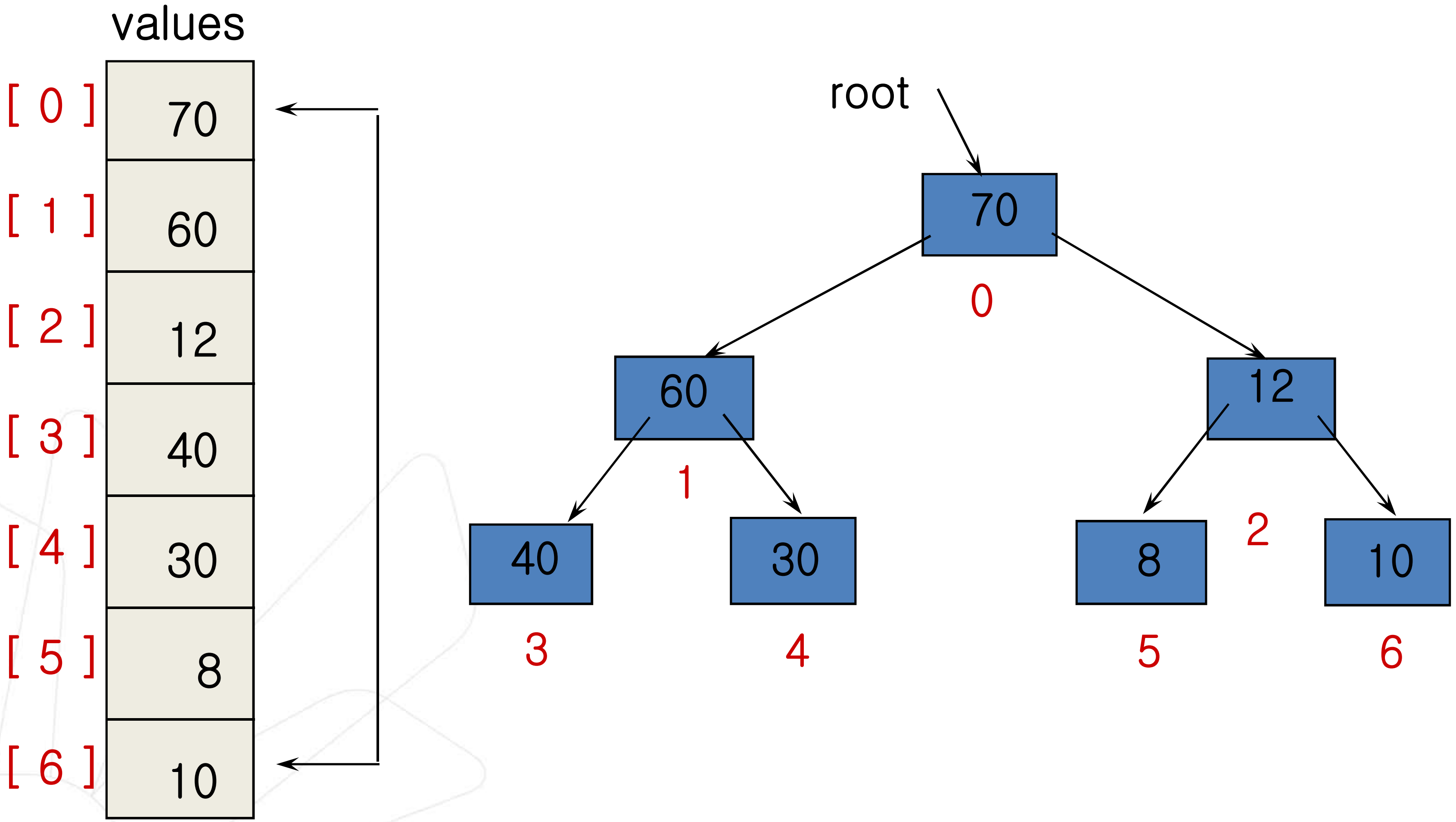


	values
[0]	70
[1]	60
[2]	12
[3]	40
[4]	30
[5]	8
[6]	10

root
70
0
60
1
12
2
40
3
30
4
8
5
10
6

Heap Sort

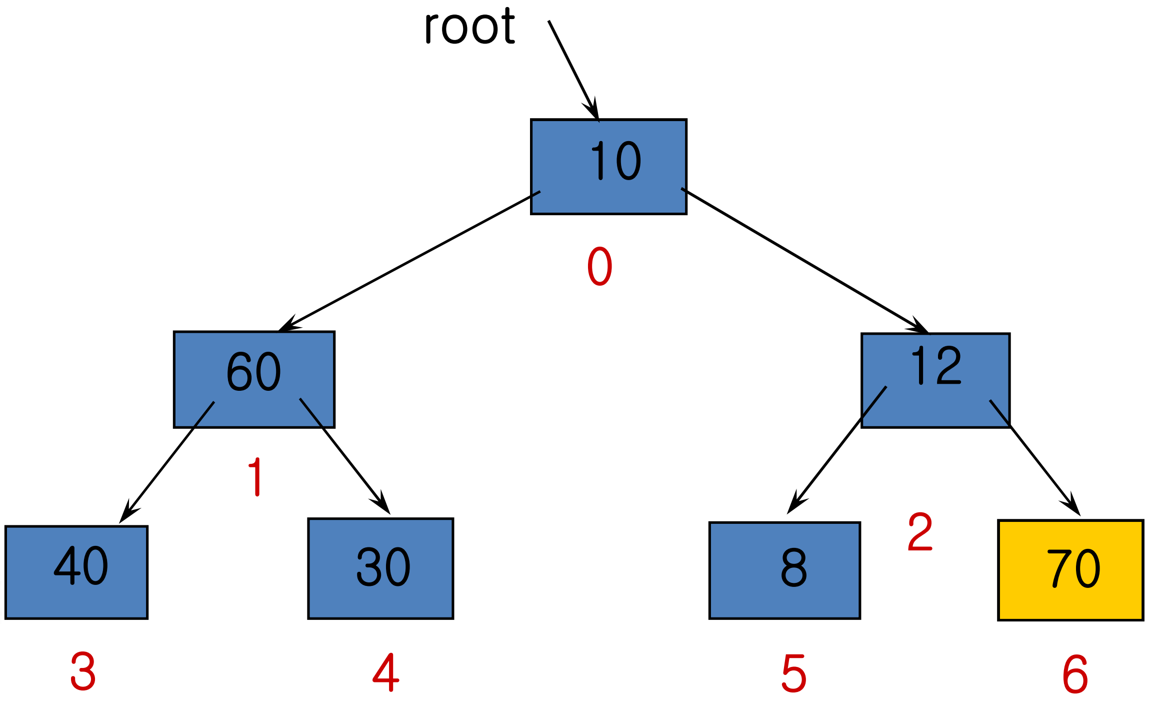
- **Swap root element into last place in unsorted array**


values
[0]
70
[1]
60
[2]
12
[3]
40
[4]
30
[5]
8
[6]
10
root
70
0
60
1
12
2
40
3
30
4
8
5
10
6

Heap Sort

- **After swapping root element into its place**

	values
[0]	10
[1]	60
[2]	12
[3]	40
[4]	30
[5]	8
[6]	70

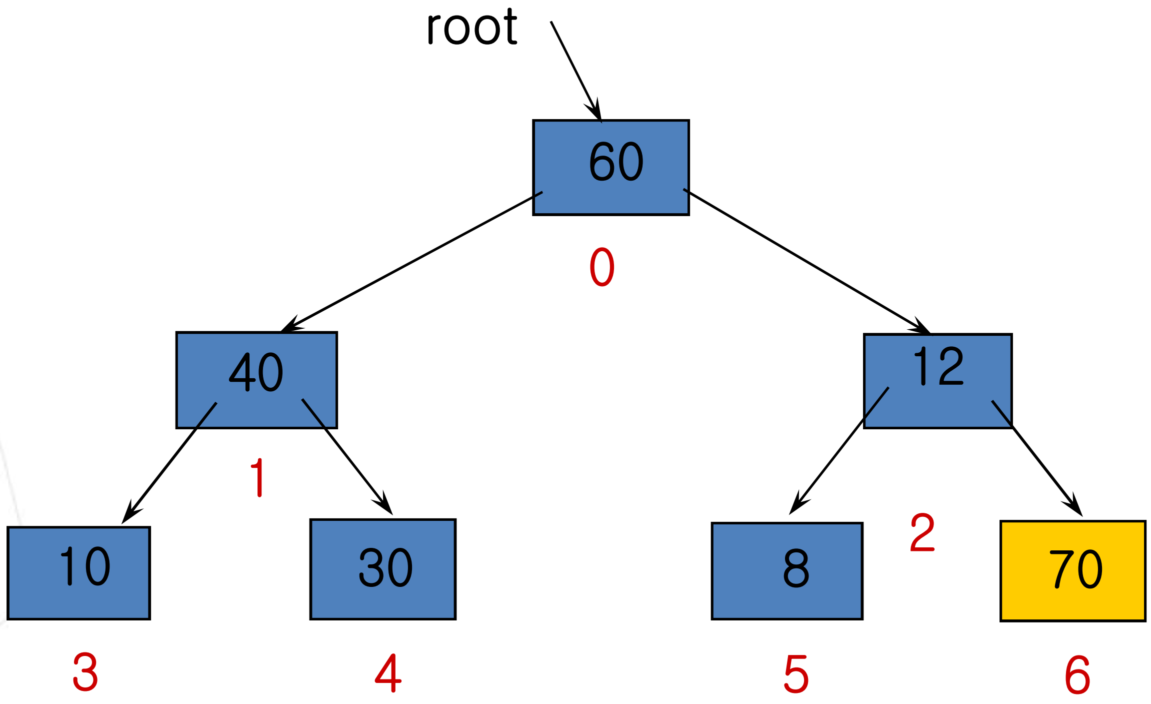

root
10
0
60
1
12
2
40
3
30
4
8
5
70
6

NO NEED TO CONSIDER AGAIN

Heap Sort

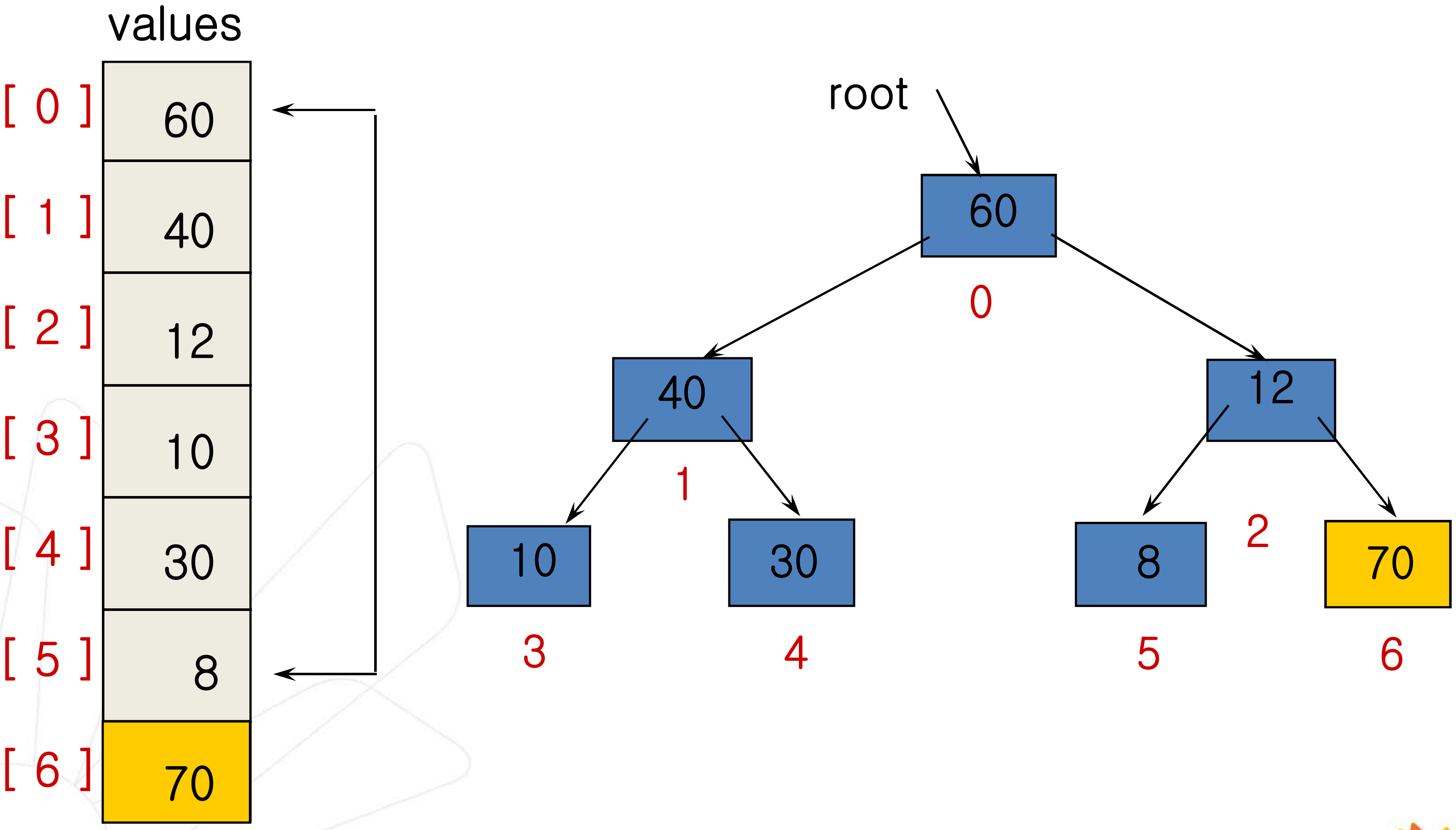
- **After reheaping remaining unsorted elements**

	values
[0]	60
[1]	40
[2]	12
[3]	10
[4]	30
[5]	8
[6]	70


root
60
0
40
1
12
2
10
3
30
4
8
5
70
6

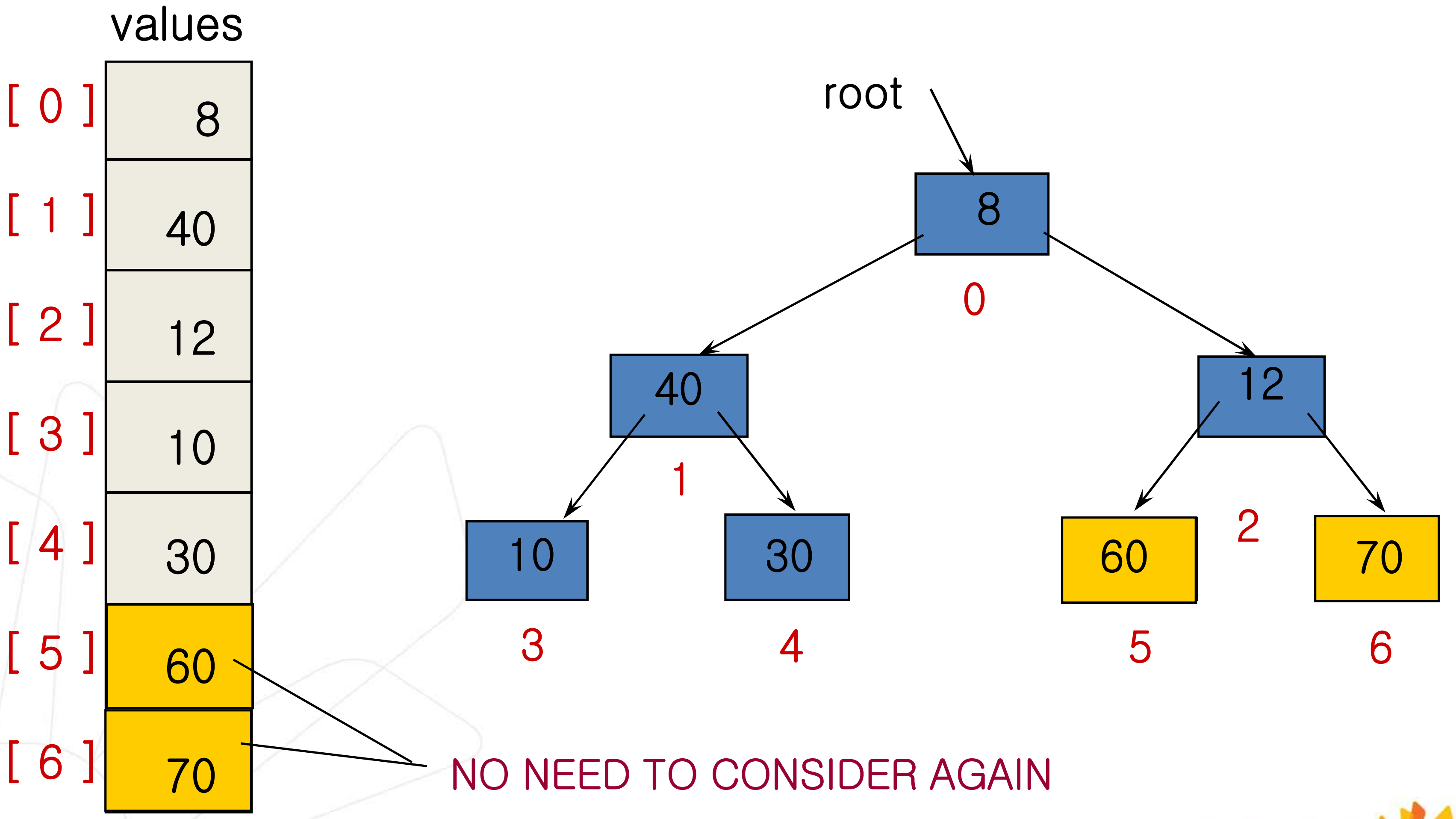
Heap Sort

- **Swap root element into last place in unsorted array**


values
[0]
60
[1]
40
[2]
12
[3]
10
[4]
30
[5]
8
[6]
70
root
60
0
40
1
12
2
10
3
30
4
8
5
70
6

Heap Sort

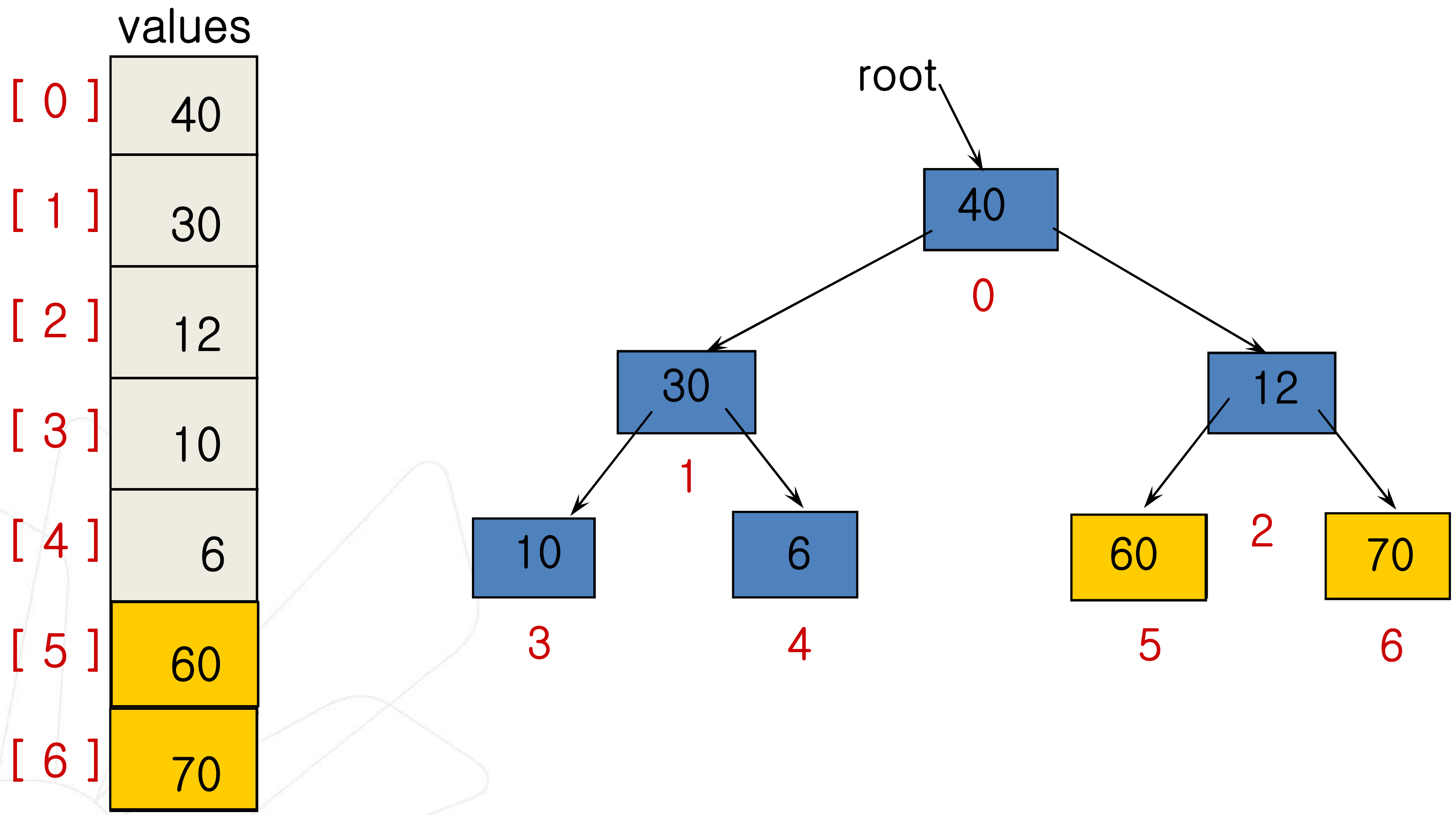
- **After swapping root element into its place**


values
[0]
8
[1]
40
[2]
12
[3]
10
[4]
30
[5]
60
[6]
70
root
8
0
40
1
12
2
10
3
30
4
60
5
70
6

NO NEED TO CONSIDER AGAIN

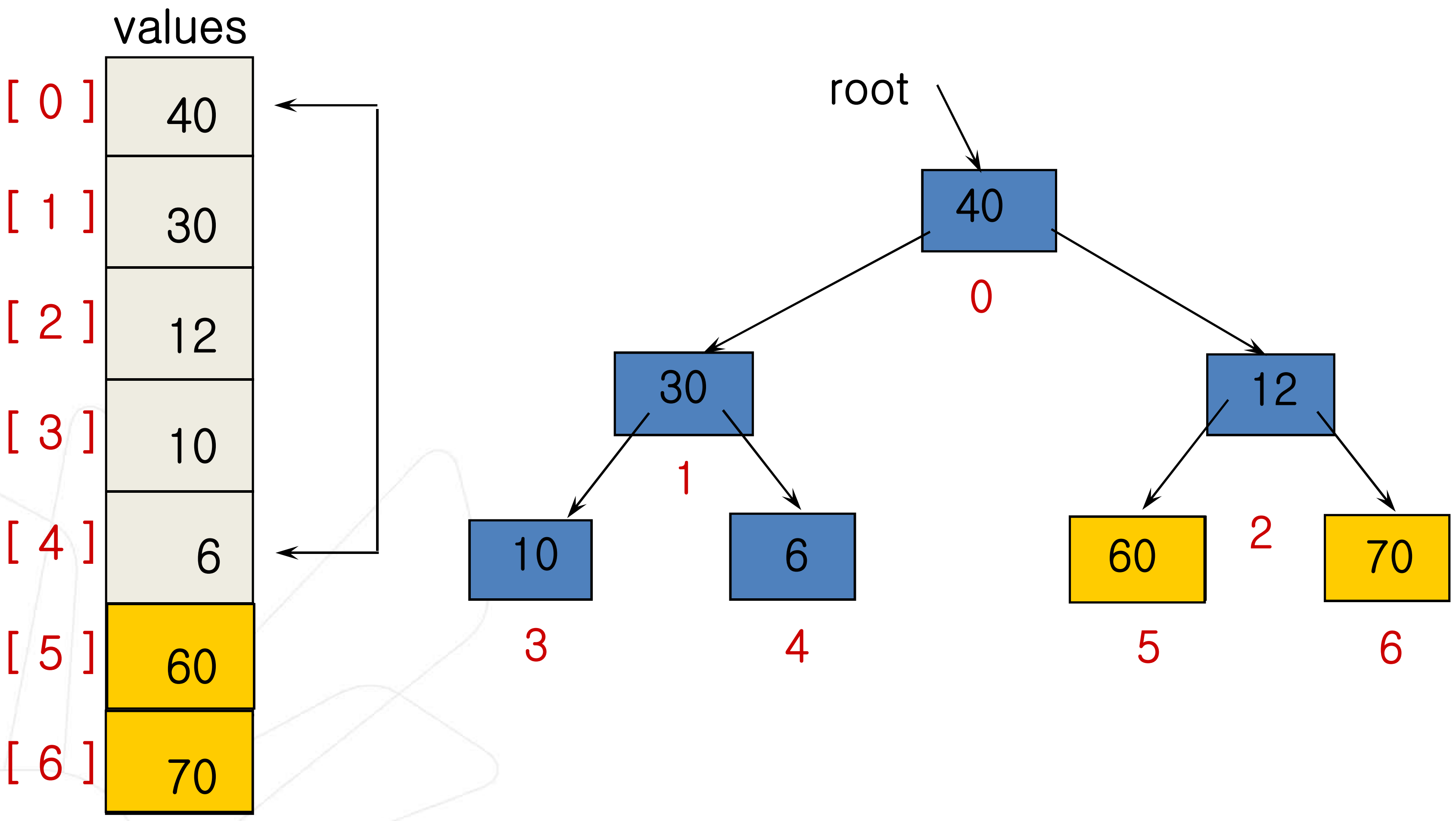
Heap Sort

- **After reheaping remaining unsorted elements**


values
[0]
40
[1]
30
[2]
12
[3]
10
[4]
6
[5]
60
[6]
70
root
40
0
30
1
12
2
10
3
6
4
60
5
70
6

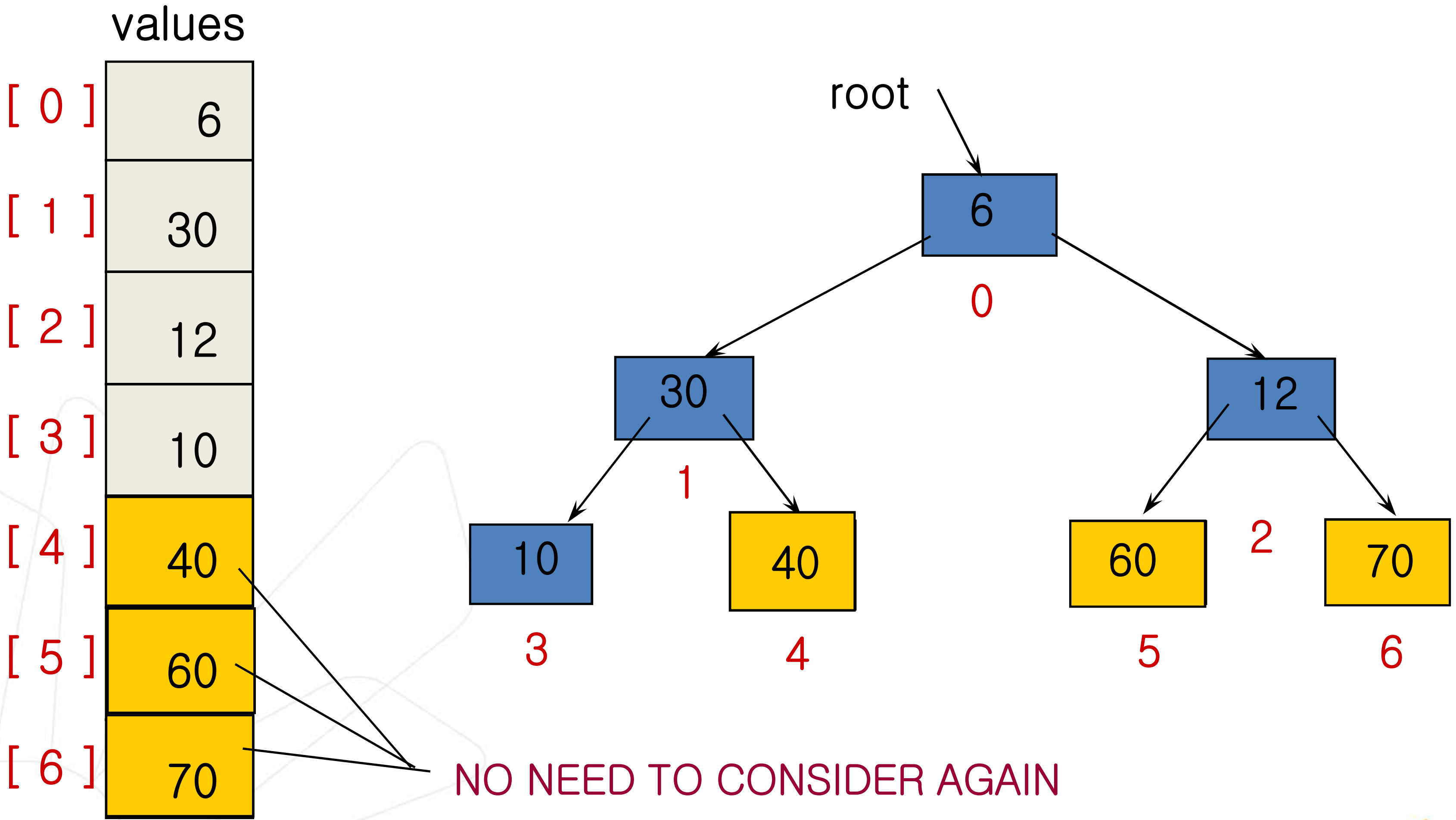
Heap Sort

- Swap root element into last place in unsorted array


values
[0]
40
[1]
30
[2]
12
[3]
10
[4]
6
[5]
60
[6]
70
root
40
0
30
1
12
2
10
3
6
4
60
5
70
6

Heap Sort

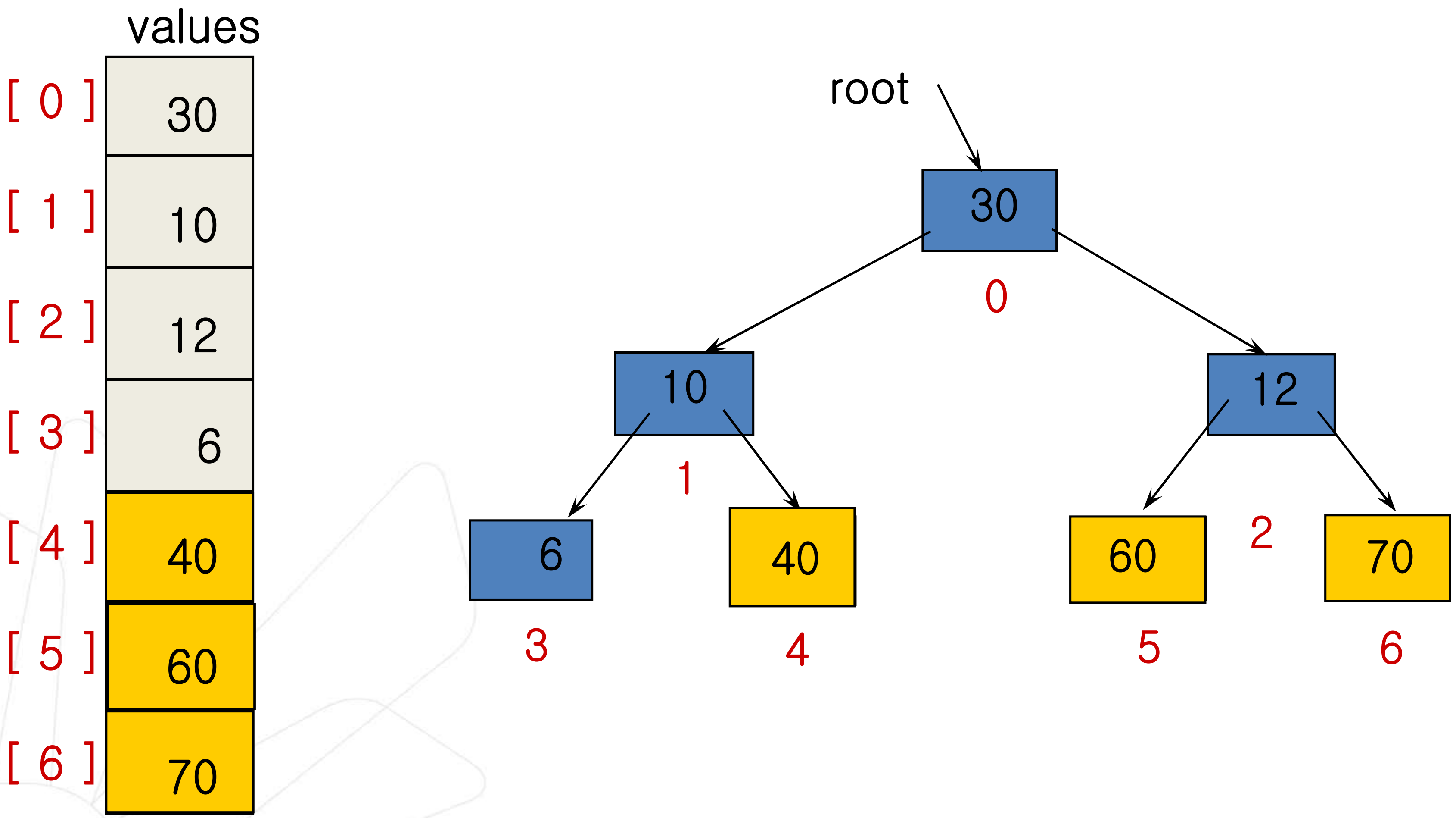
- **After swapping root element into its place**


values
[0] 6
[1] 30
[2] 12
[3] 10
[4] 40
[5] 60
[6] 70
root
6
0
30
1
12
2
10
3
40
4
60
5
70
6

NO NEED TO CONSIDER AGAIN

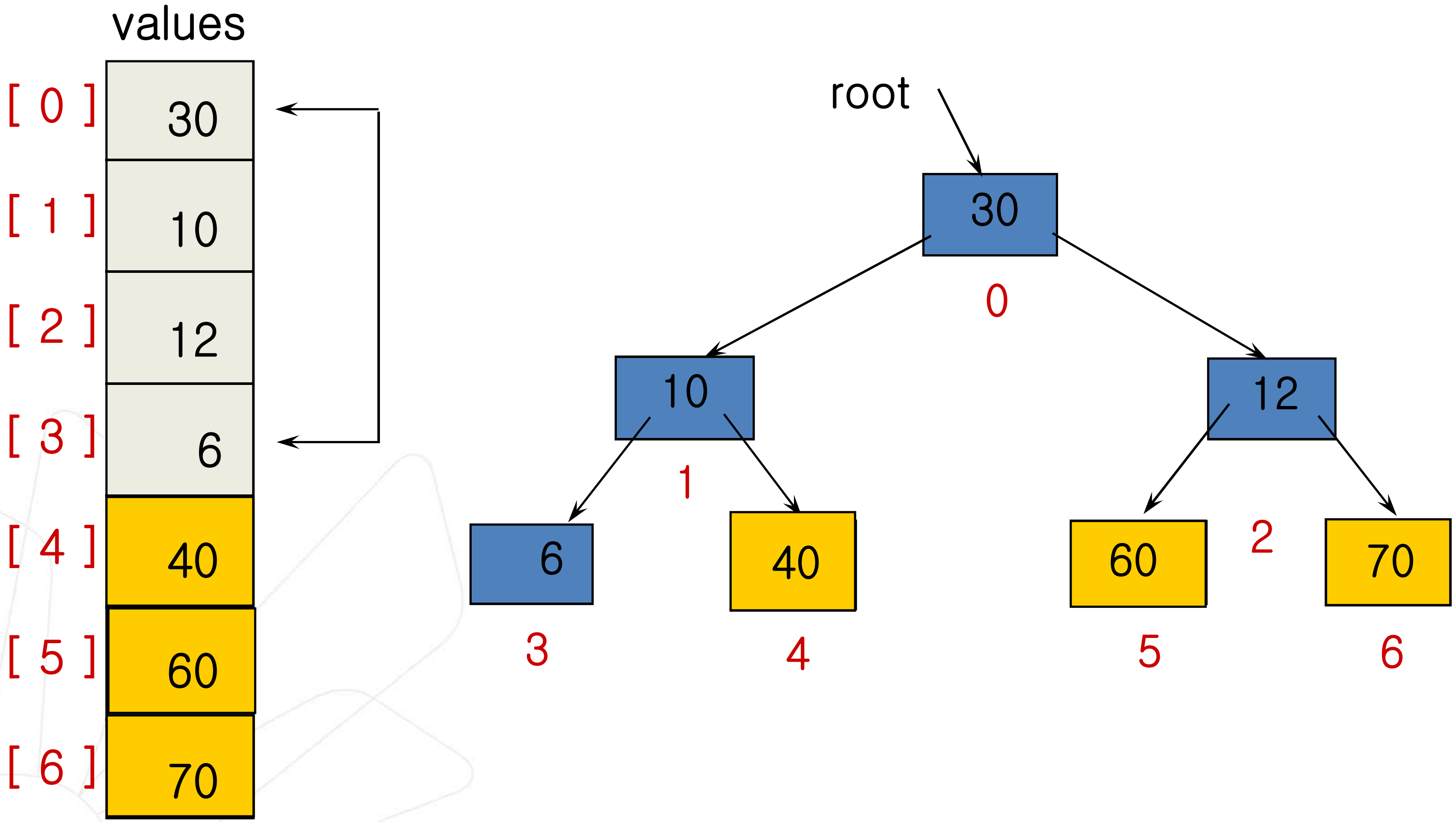
Heap Sort

- **After reheaping remaining unsorted elements**


values
[0] 30
[1] 10
[2] 12
[3] 6
[4] 40
[5] 60
[6] 70
root
30
0
10
1
12
2
6
3
40
4
60
5
70
6

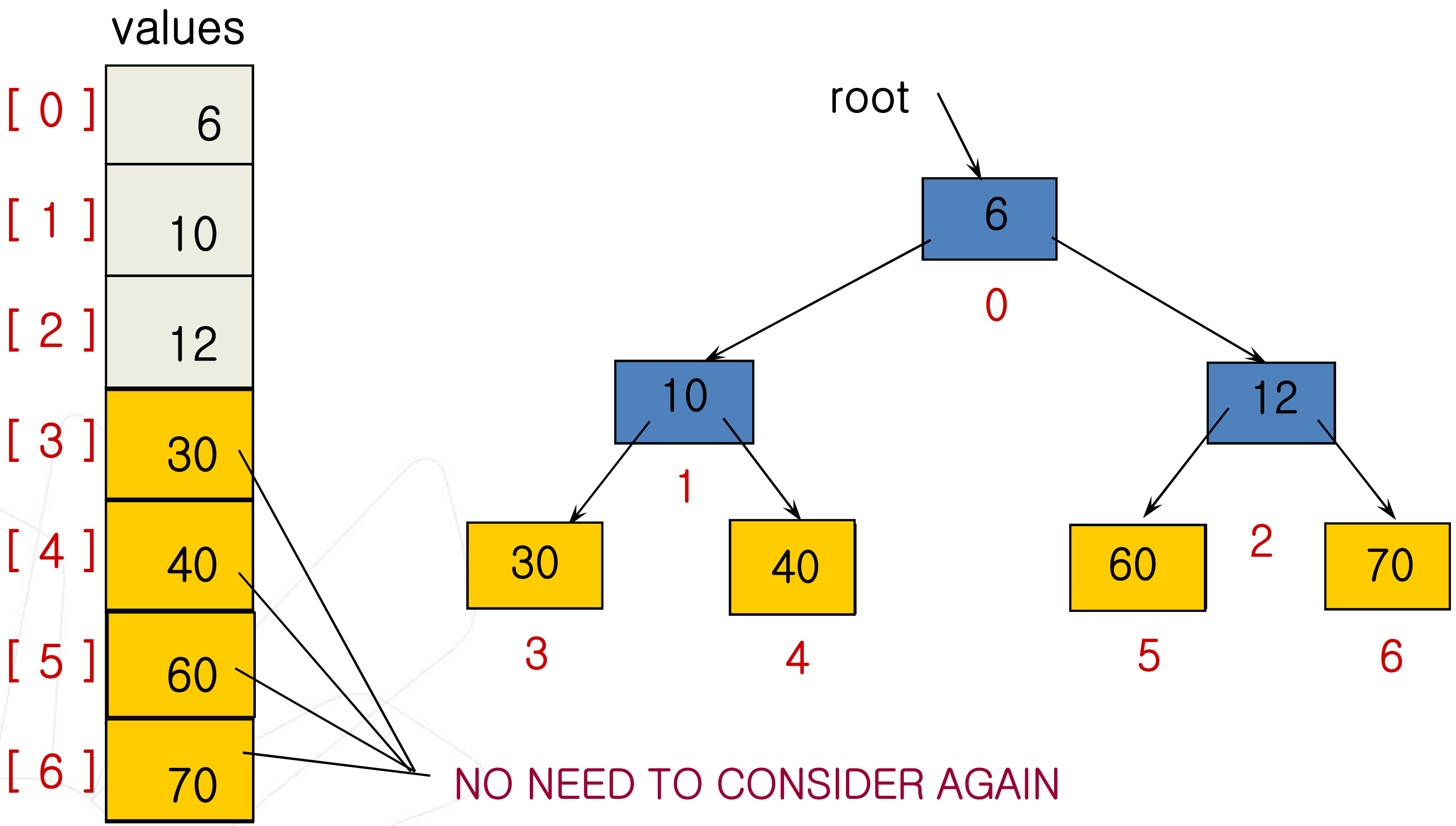
Heap Sort

- **Swap root element into last place in unsorted array**


values
[0]
30
[1]
10
[2]
12
[3]
6
[4]
40
[5]
60
[6]
70
root
30
0
10
1
12
2
6
3
40
4
60
5
70
6

Heap Sort

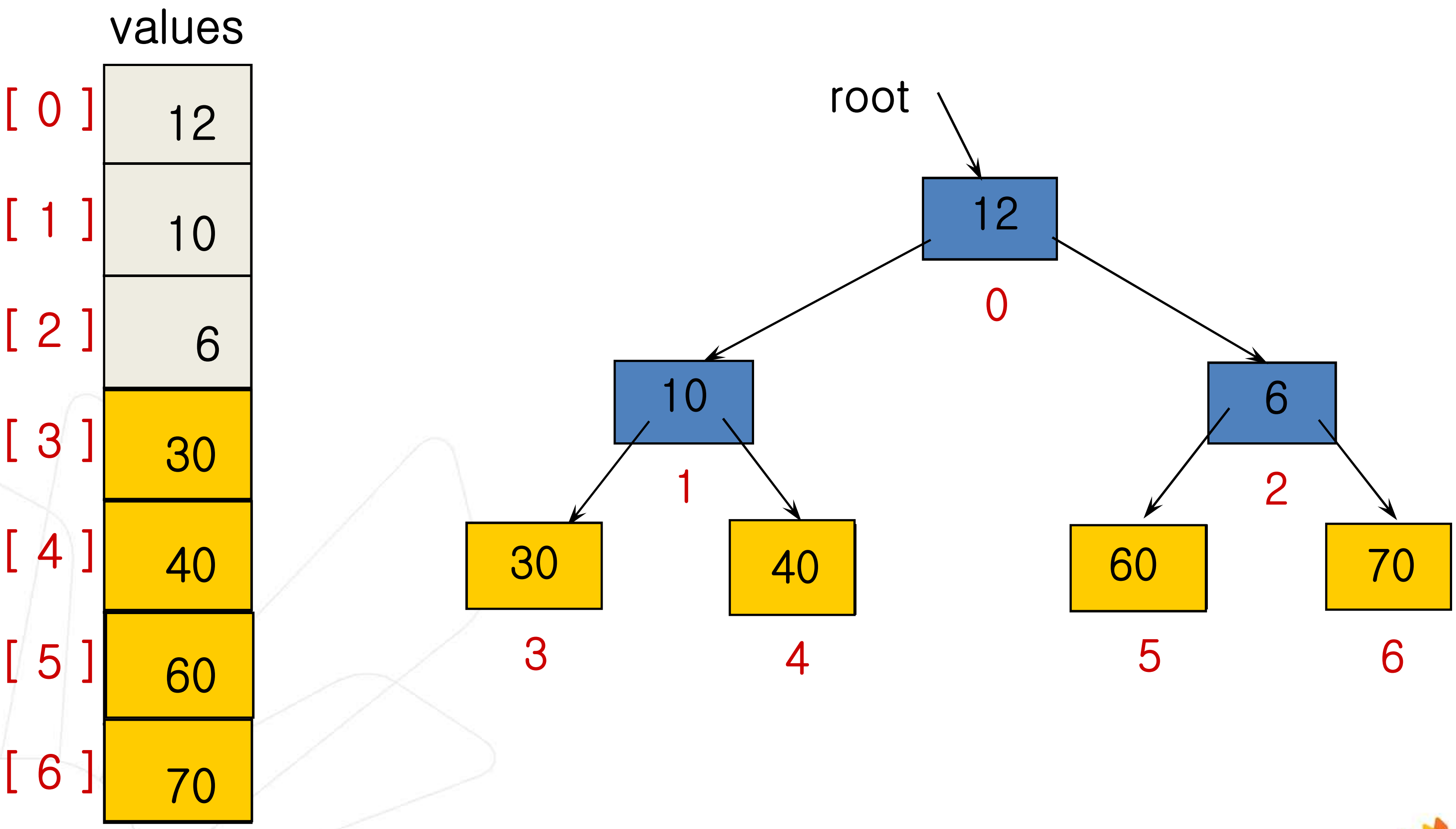
- **After swapping root element into its place**


values
[0]
6
[1]
10
[2]
12
[3]
30
[4]
40
[5]
60
[6]
70
root
6
0
10
1
12
2
30
3
40
4
60
5
70
6

NO NEED TO CONSIDER AGAIN

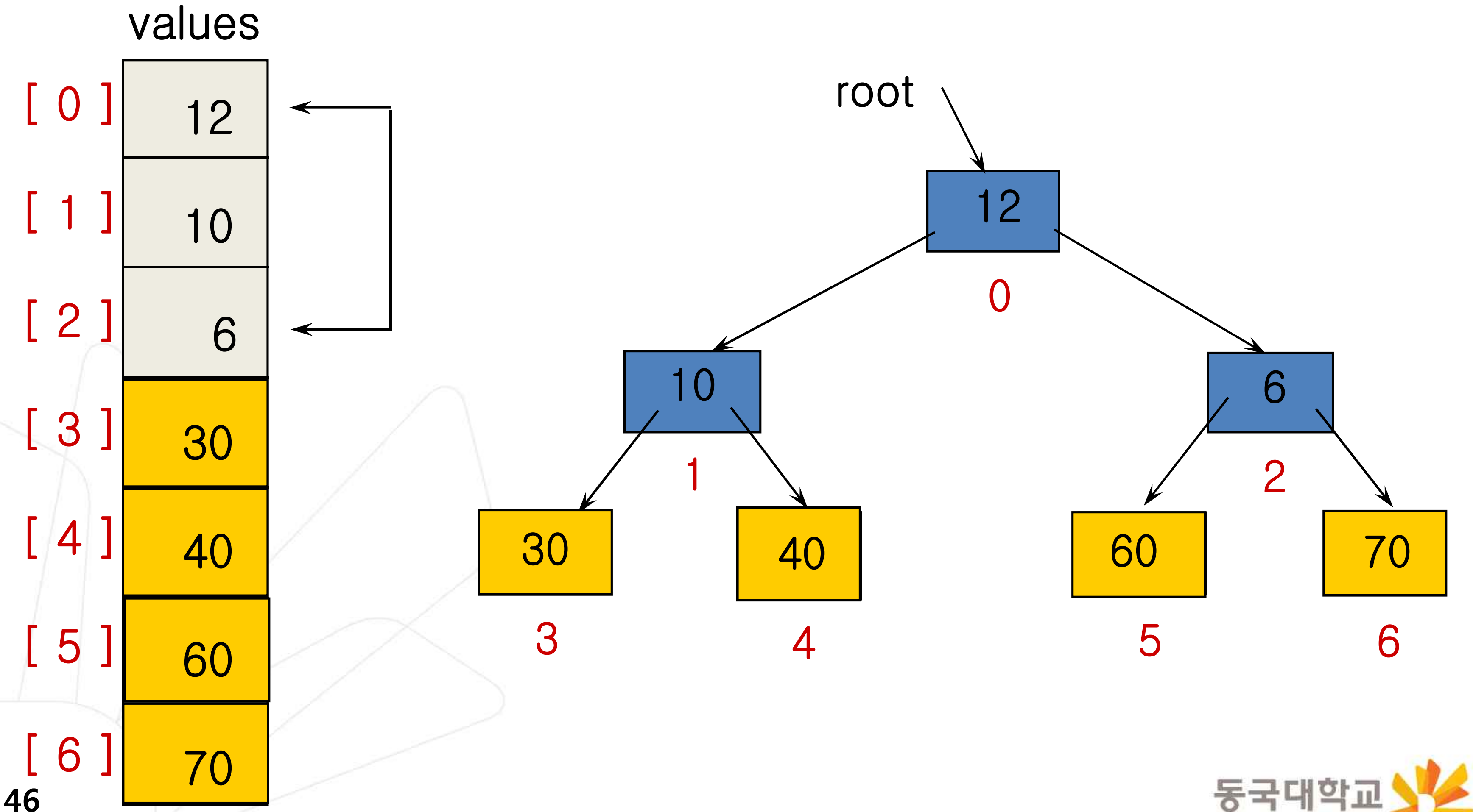
Heap Sort

- **After reheaping remaining unsorted elements**


values
[0] 12
[1] 10
[2] 6
[3] 30
[4] 40
[5] 60
[6] 70
root
12
0
10
1
6
2
30
3
40
4
60
5
70
6

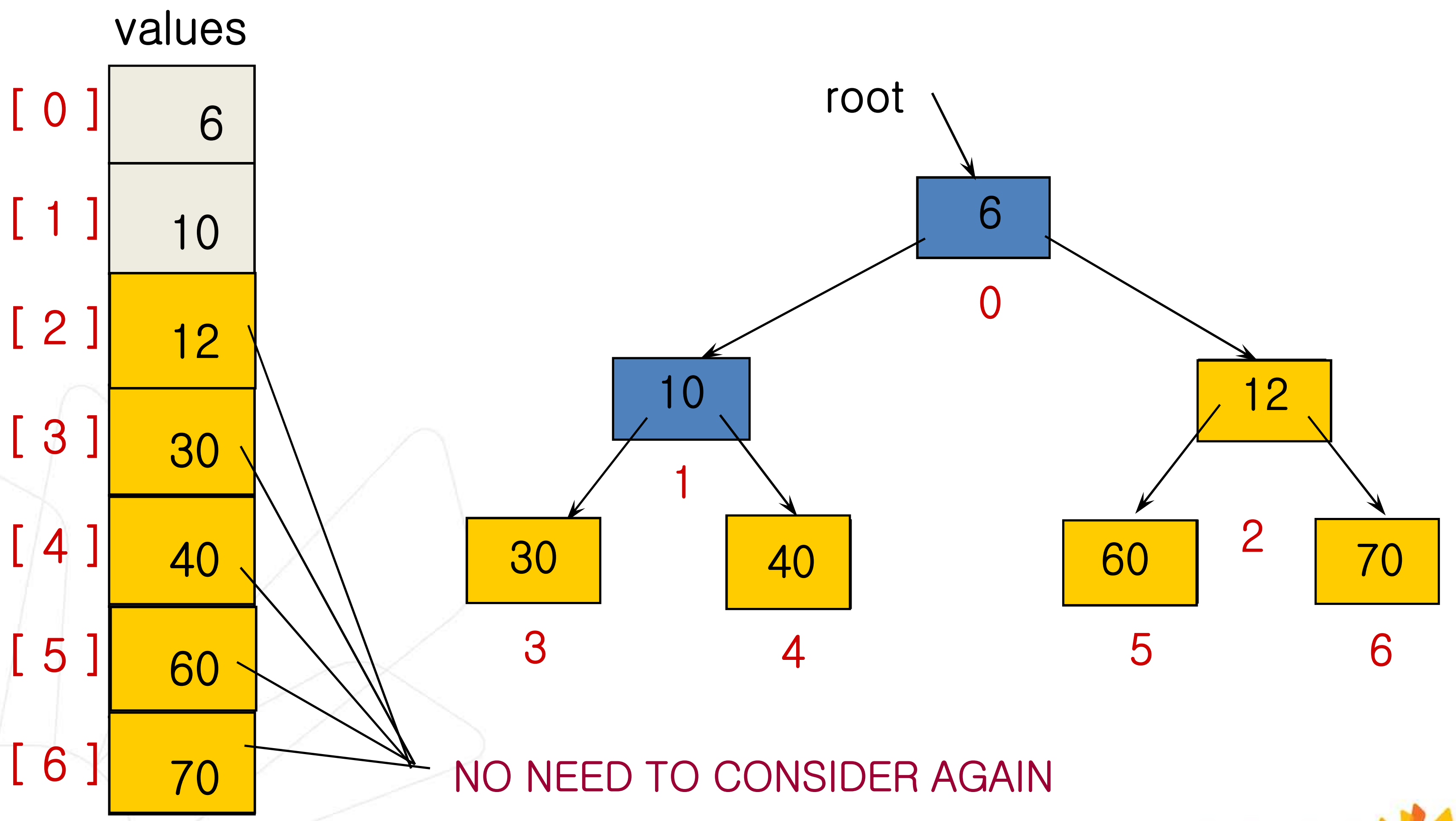
Heap Sort

- **Swap root element into last place in unsorted array**


values
[0] 12
[1] 10
[2] 6
[3] 30
[4] 40
[5] 60
[6] 70
root
12
0
10
1
6
2
30
3
40
4
60
5
70
6

Heap Sort

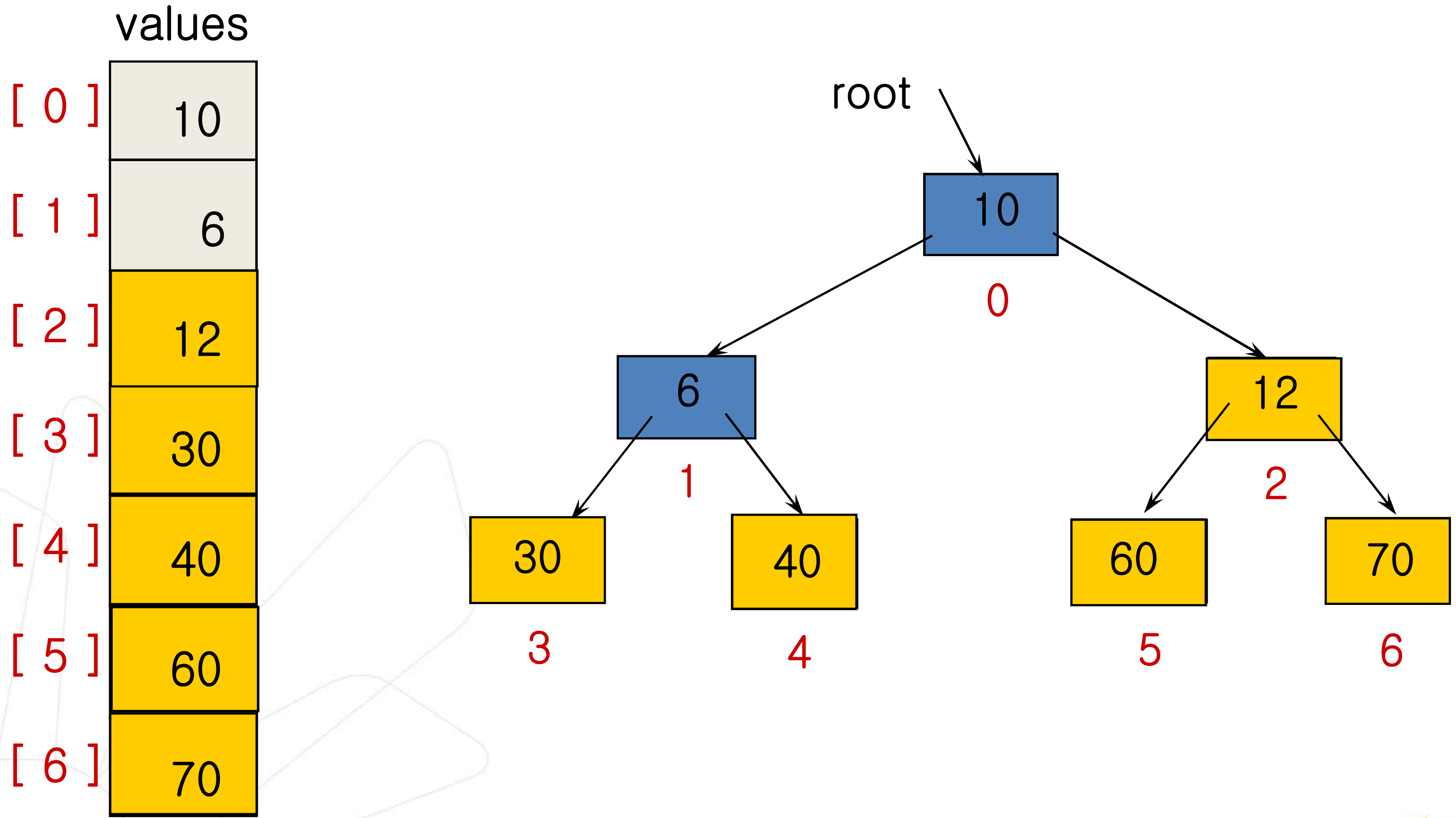
- **After swapping root element into its place**


values
[0]
6
[1]
10
[2]
12
[3]
30
[4]
40
[5]
60
[6]
70
root
6
0
10
1
12
2
30
3
40
4
60
5
70
6

NO NEED TO CONSIDER AGAIN

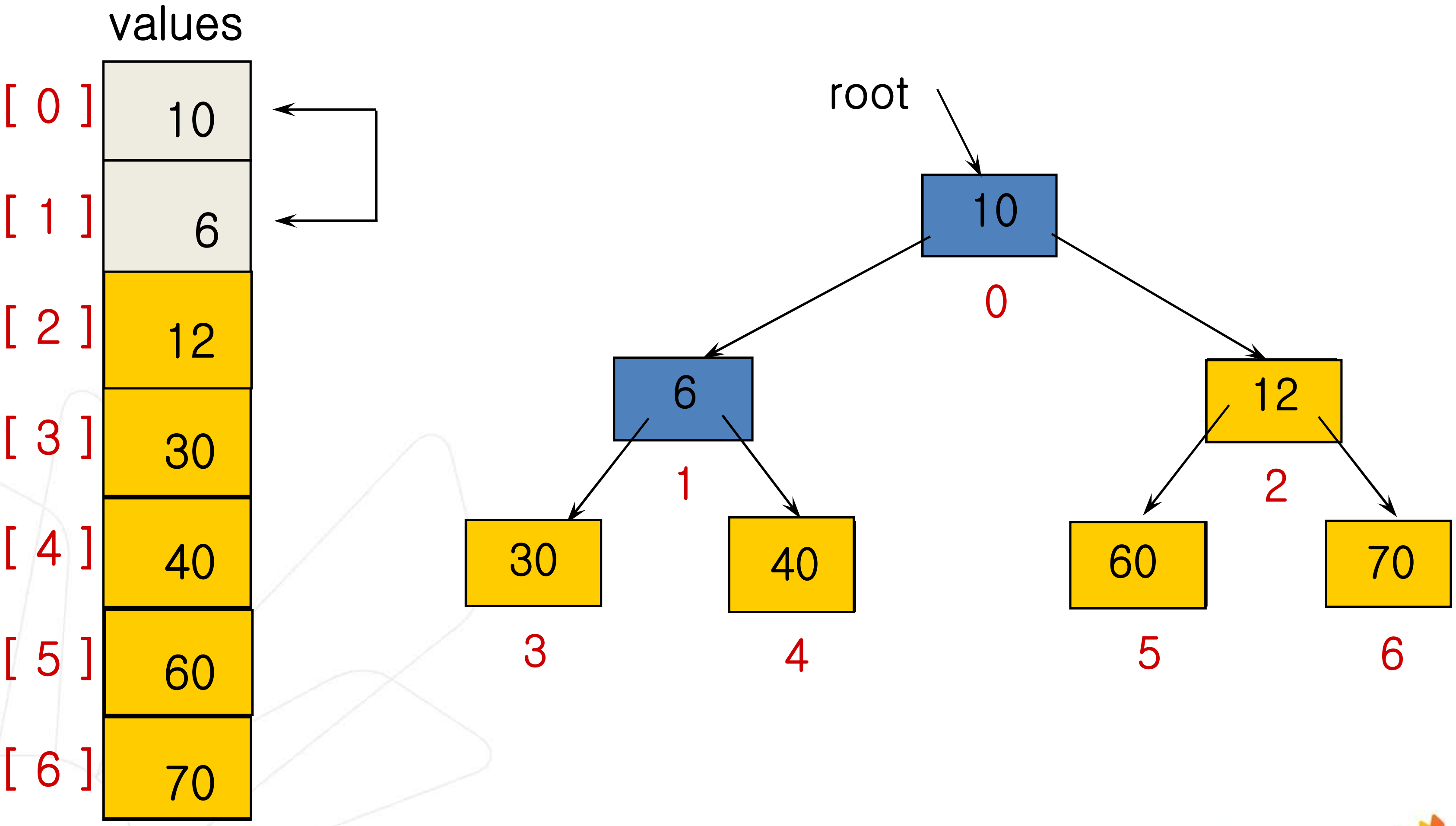
Heap Sort

- **After reheaping remaining unsorted elements**


values
[0] 10
[1] 6
[2] 12
[3] 30
[4] 40
[5] 60
[6] 70
root
10
0
6
1
12
2
30
3
40
4
60
5
70
6

Heap Sort

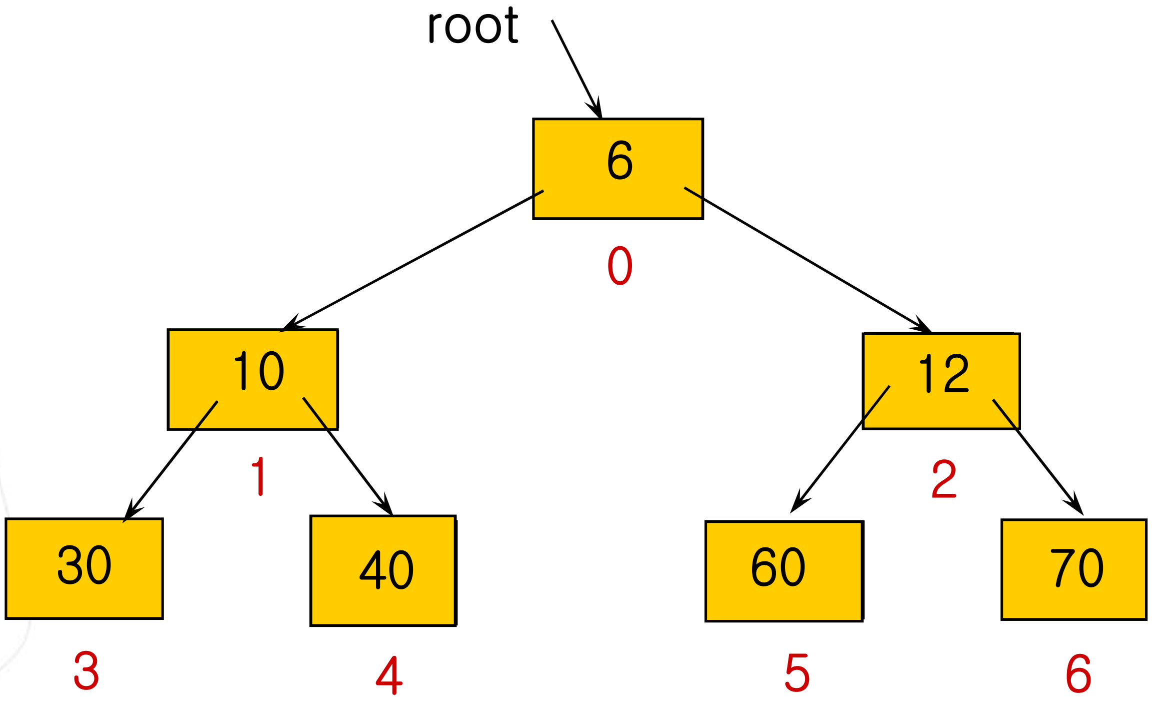
- **Swap root element into last place in unsorted array**


values
[0]
10
[1]
6
[2]
12
[3]
30
[4]
40
[5]
60
[6]
70
root
10
0
6
1
12
2
30
3
40
4
60
5
70
6

Heap Sort

- **After swapping root element into its place**

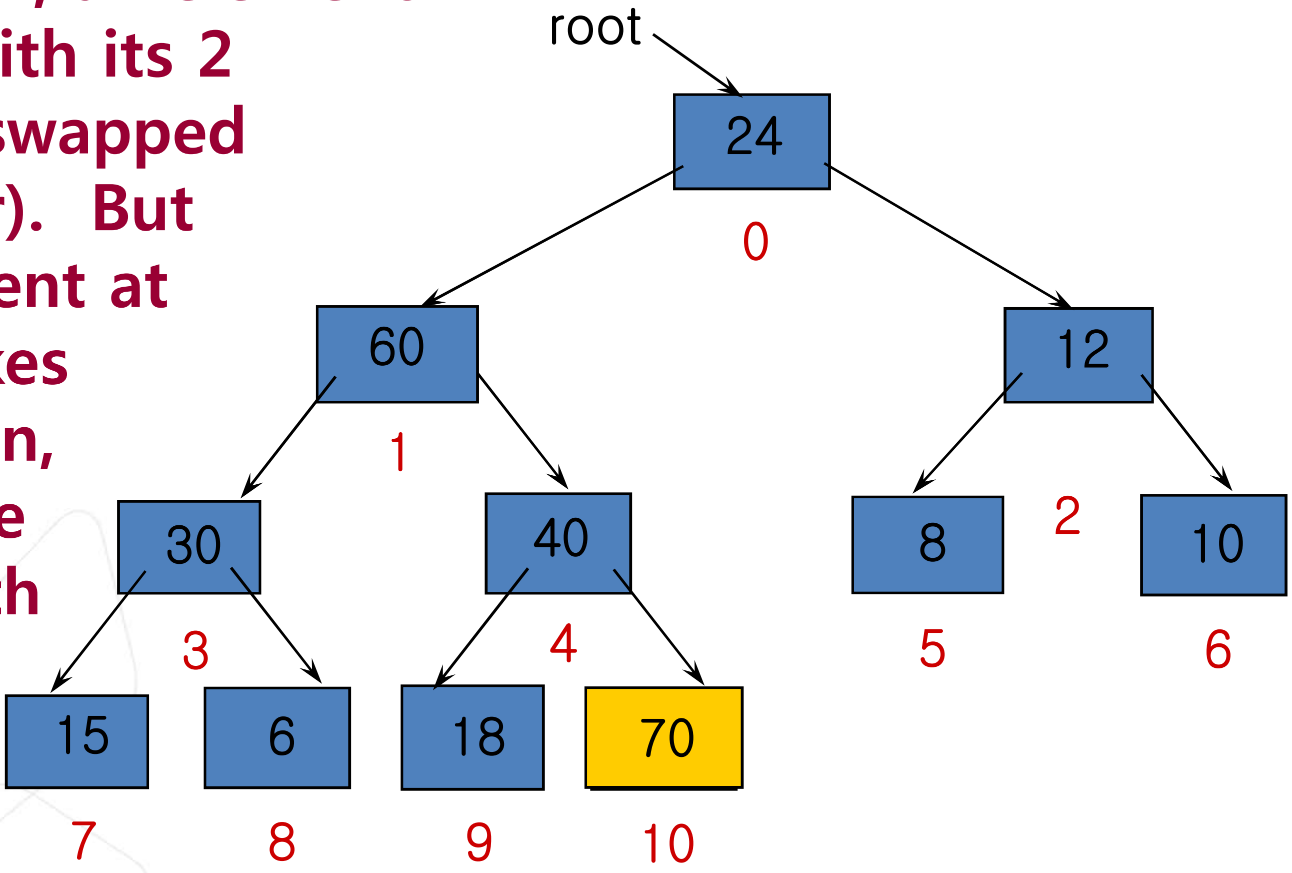
	values
[0]	6
[1]	10
[2]	12
[3]	30
[4]	40
[5]	60
[6]	70


root
6
0
10
1
12
2
30
3
40
4
60
5
70
6

ALL ELEMENTS ARE SORTED

Heap Sort: How many comparisons?

In reheap down, an element is compared with its 2 children (and swapped with the larger). But only one element at each level makes this comparison, and a complete binary tree with N nodes has only O($\log_2 N$) levels.


root
24
0
60
1
12
2
30
3
40
4
8
5
10
6
15
7
6
8
18
9
70
10

동국대학교 dongguk university

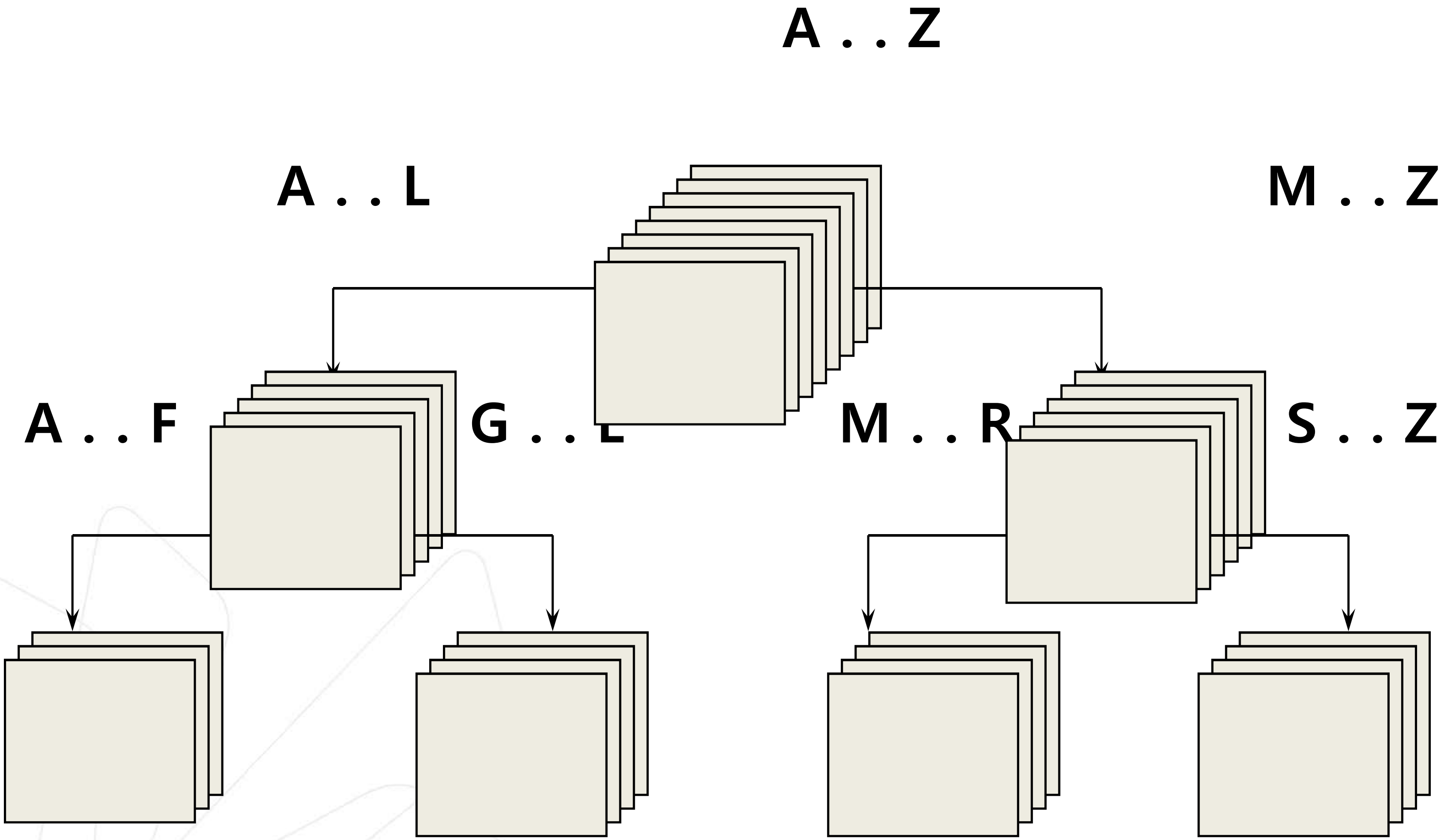
Heap Sort of N elements: How many comparisons?

(N/2) * O(log N) compares to create original heap

(N-1) * O(log N) compares for the sorting loop

= O (N * log N) compares total

Using quick sort algorithm



A . . Z

A . . L

M . . Z

A . . F

G . . L

M . . R

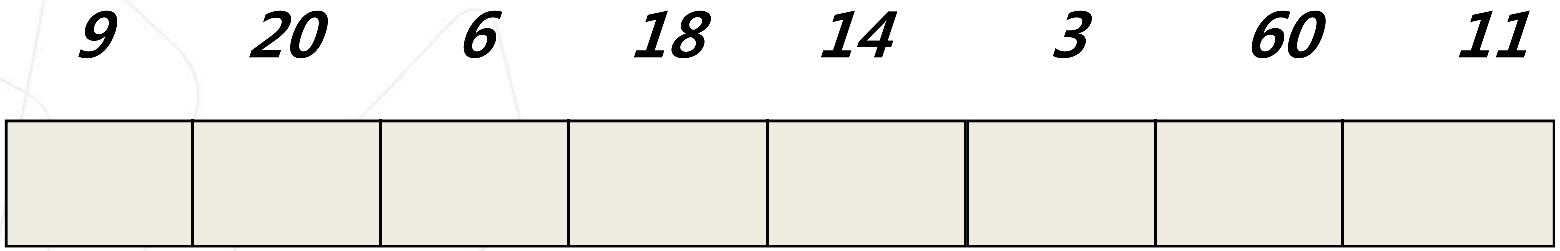
S . . Z

```
// Recursive quick sort algorithm

template <class  ItemType >
void  QuickSort  ( ItemType  values[ ] ,  int  first ,  int  last )


// Pre:   first <= last
// Post: Sorts array values[ first .  . last ] into ascending order
{
  if  ( first < last )                              //  general case

  {       int  splitPoint ;

          Split ( values, first, last, splitPoint ) ;

          // values [ first ] . . values[splitPoint - 1 ] <= splitVal
          // values  [ splitPoint ] = splitVal
          // values [ splitPoint + 1 ] . . values[ last ] > splitVal

          QuickSort( values,  first,  splitPoint - 1 ) ;
          QuickSort( values,  splitPoint + 1,  last );
  }
} ;
```

Before call to function Split

splitVal = 9

GOAL: place splitVal in its proper position with
all values less than or equal to splitVal on its left
and all larger values on its right



values[first]

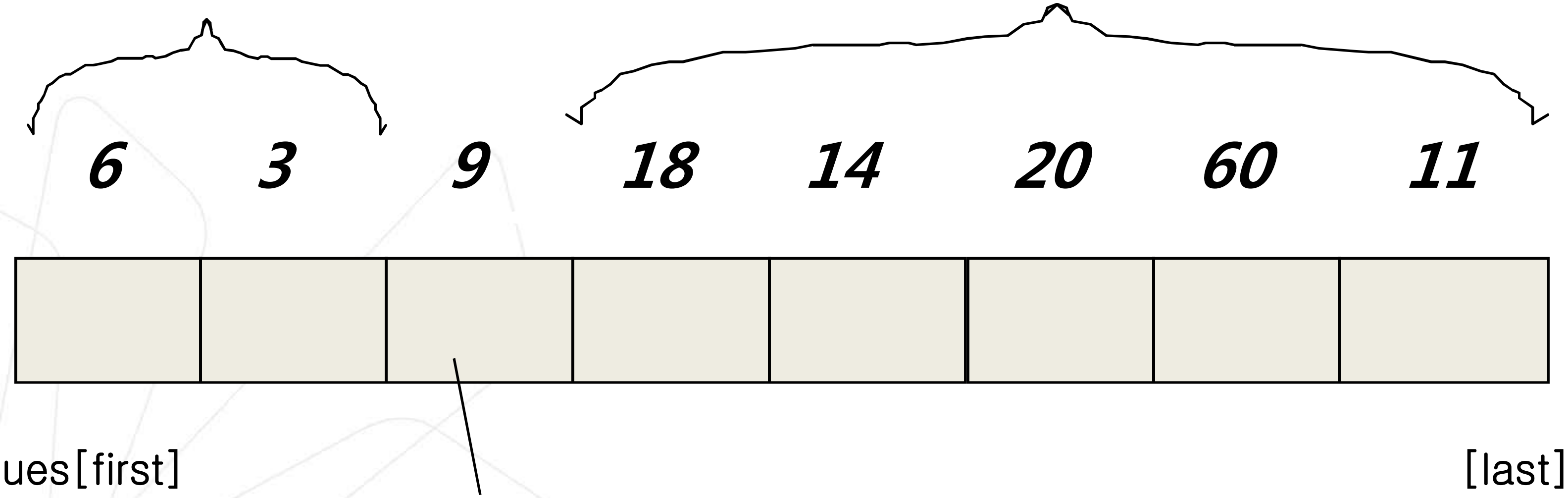
[last]

After call to function Split

splitVal = 9

smaller values
in left part

larger values
in right part



values[first]

[last]

splitVal in correct position

Quick Sort of N elements: How many comparisons

N	For first call, when each of N elements is compared to the split value
2 * N/2	For the next pair of calls, when N/2 elements in each “half” of the original array are compared to their own split values.
4 * N/4	For the four calls when N/4 elements in each “quarter” of original array are compared to their own split values.

.

.

.

HOW MANY SPLITS CAN OCCUR?

Quick Sort of N elements: How many splits can occu

It depends on the order of the original array elements!

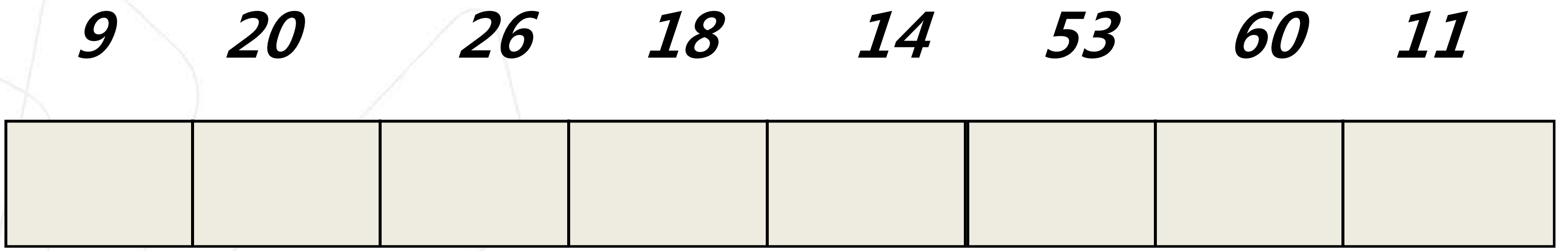
If each split divides the subarray approximately in half, there will be only $\log_2 N$ splits, and QuickSort is $O(N*\log_2 N)$.

But, if the original array was sorted to begin with, the recursive calls will split up the array into parts of unequal length, with one part empty, and the other part containing all the rest of the array except for split value itself. In this case, there can be as many as N-1 splits, and QuickSort is $O(N^2)$.

Before call to function Split

splitVal = 9

GOAL: place splitVal in its proper position with all values less than or equal to splitVal on its left and all larger values on its right

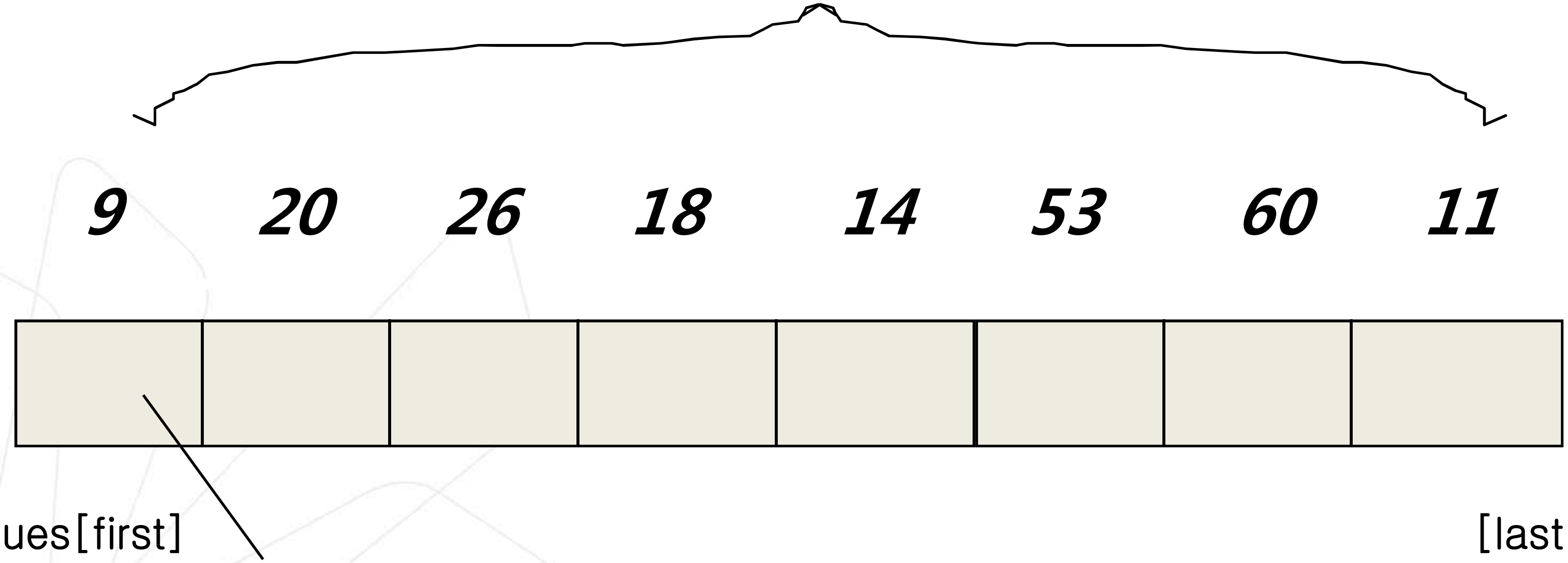


9 20 26 18 14 53 60 11

values[first] [last]

After call to function Split

splitVal = 9



no smaller values
empty left part

larger values
in right part with N-1 elements

values[first]

[last]

splitVal in correct position

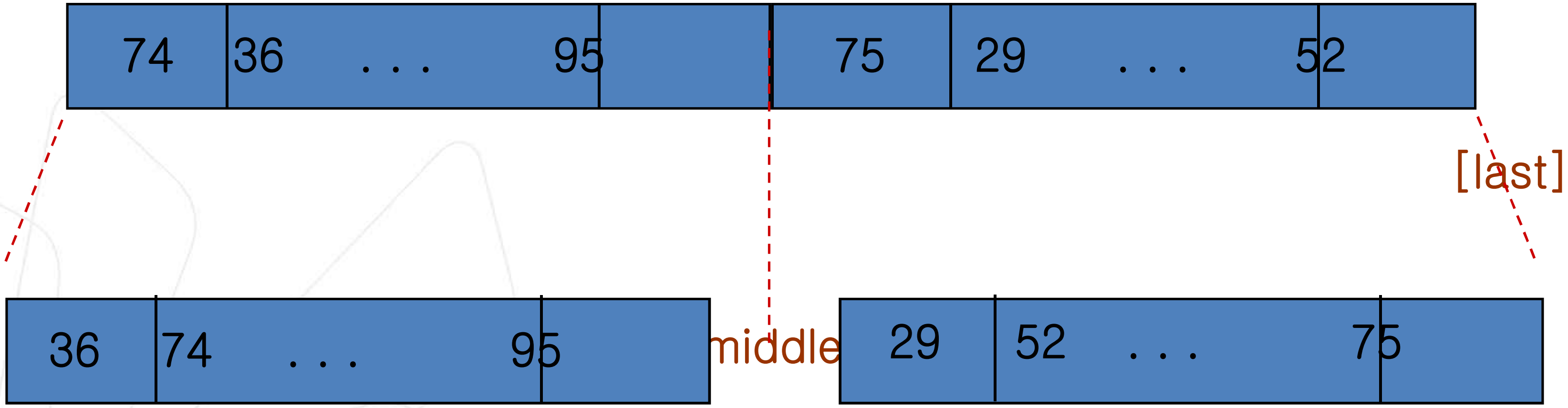
Merge Sort Algorithm

- Cut the array in half.

Sort the left half.

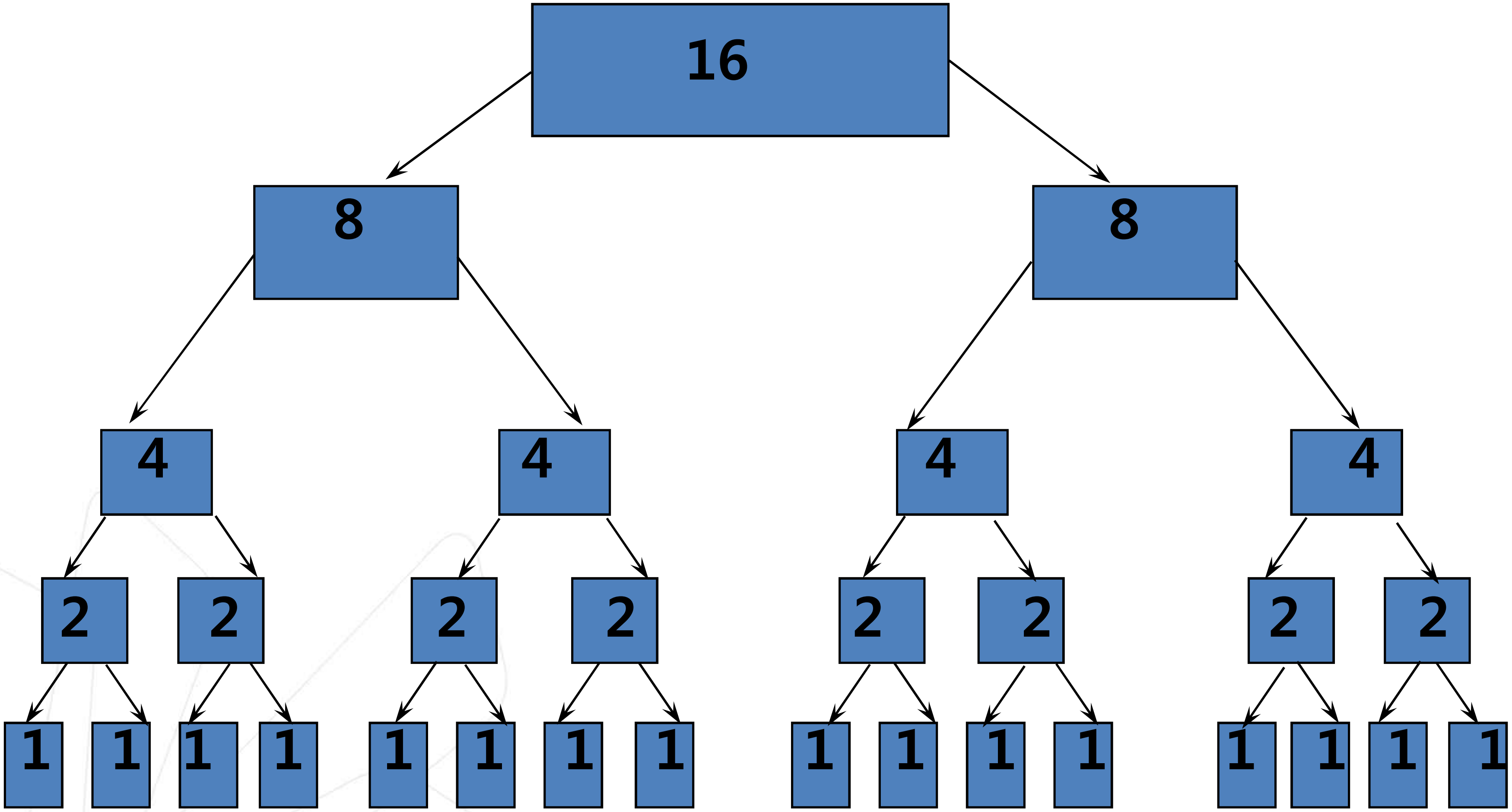
Sort the right half.

Merge the two sorted halves into one sorted array.


74
36
. . .
95
75
29
. . .
52
[last]
36
74
. . .
95
middle
29
52
. . .
75

```
// Recursive merge sort algorithm

template <class  ItemType >
void  MergeSort  ( ItemType  values[ ] ,  int  first ,  int  last
//  Pre:   first <= last
//  Post: Array values[ first . . last ] sorted into ascending
   order.
{
   if  ( first < last )                          //  general case
   {        int  middle = ( first  +  last ) / 2  ;
            MergeSort ( values, first, middle ) ;
            MergeSort( values,  middle + 1, last ) ;
            // now  merge two subarrays
            // values  [ first . . . middle ] with
            // values [ middle + 1,  . . . last ].
            Merge( values,  first, middle, middle + 1, last ) ;
   }
}
```

Using Merge Sort Algorithm with N = 16


16
8
8
4
4
4
4
2
2
2
2
2
2
2
2
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1

Merge Sort of N elements: How many comparisons?

- **The entire array can be subdivided into halves only log2N times.**

- **Each time it is subdivided, function Merge is called to re-combine the halves. Function Merge uses a temporary array to store the merged elements. Merging is O(N) because it compares each element in the subarrays.**

- **Copying elements back from the temporary array to the values array is also O(N).**

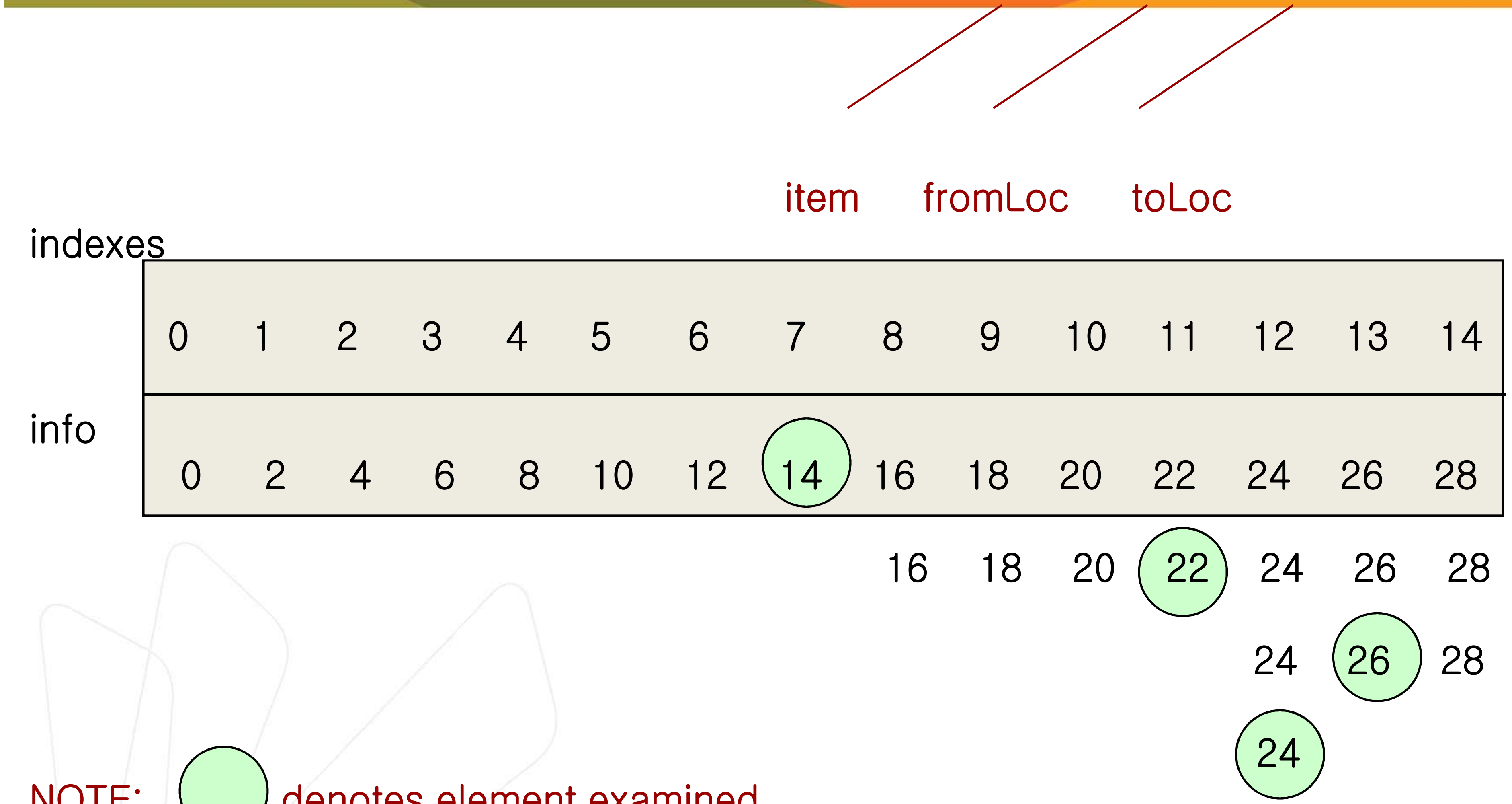
- **MERGE SORT IS O(N*log2N).**

Function `BinarySearch( )`

- `BinarySearch` takes sorted array `info`, and two subscripts, `fromLoc` and `toLoc`, and `item` as arguments. It returns false if `item` is not found in the elements `info[fromLoc…toLoc]`. Otherwise, it returns true.

- `BinarySearch` is O($\log_2 N$).

`found = BinarySearch(info, 25, 0, 14 );`



item
fromLoc
toLoc

indexes	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
info	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28

16 18 20 22 24 26 28

24 26 28

24

NOTE: ○ denotes element examined

```
template<class  ItemType>
bool  BinarySearch   (  ItemType  info[ ] ,  ItemType  item ,
                                    int   fromLoc ,    int  toLoc )
  //    Pre:   info [ fromLoc . . toLoc ] sorted in ascending order
  //    Post:   Function value =  ( item  in info [ fromLoc . . toLoc] )

{          int  mid ;
           if  ( fromLoc > toLoc )                    //  base case -- not found
                      return  false ;
           else  {
                     mid = ( fromLoc + toLoc ) / 2 ;

                  if ( info [ mid ] == item )       // base case-- found at mid

                        return  true  ;

                  else  if ( item < info [ mid ] )          //  search lower half
                        return BinarySearch ( info, item, fromLoc, mid-1 ) ;
                  else                                       // search upper half
                        return  BinarySearch( info, item, mid + 1, toLoc ) ;
            }

}
```

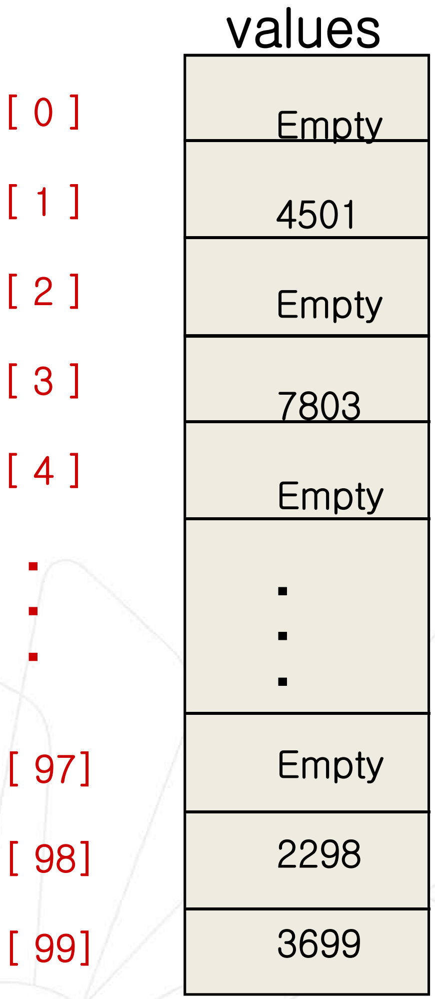
Hashing

- is a means used to order and access elements in a list quickly -- the goal is O(1) time -- by using a function of the key value to identify its location in the list.

- The function of the key value is called a hash function.

FOR EXAMPLE . . .

Using a hash function



	values
[0]	Empty
[1]	4501
[2]	Empty
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

HandyParts company makes no more than 100 different parts. But the parts all have four digit numbers.

This hash function can be used to store and retrieve parts in an array.

Hash(key) = partNum % 100

Placing elements in the array

	values
[0]	Empty
[1]	4501
[2]	Empty
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

Use the hash function

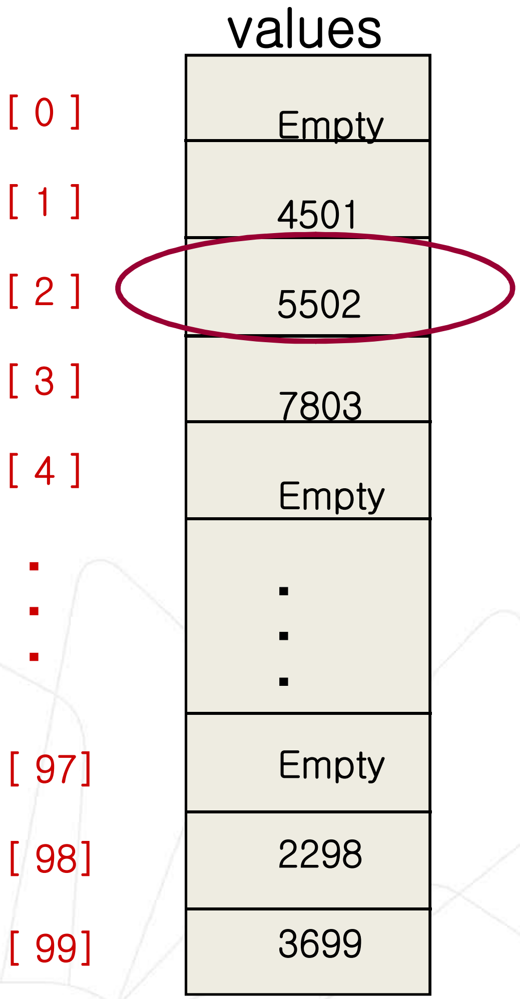
Hash(key) = partNum % 100

to place the element with

part number 5502 in the

array.

Placing elements in the array



	values
[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

Next place part number 6702 in the array.

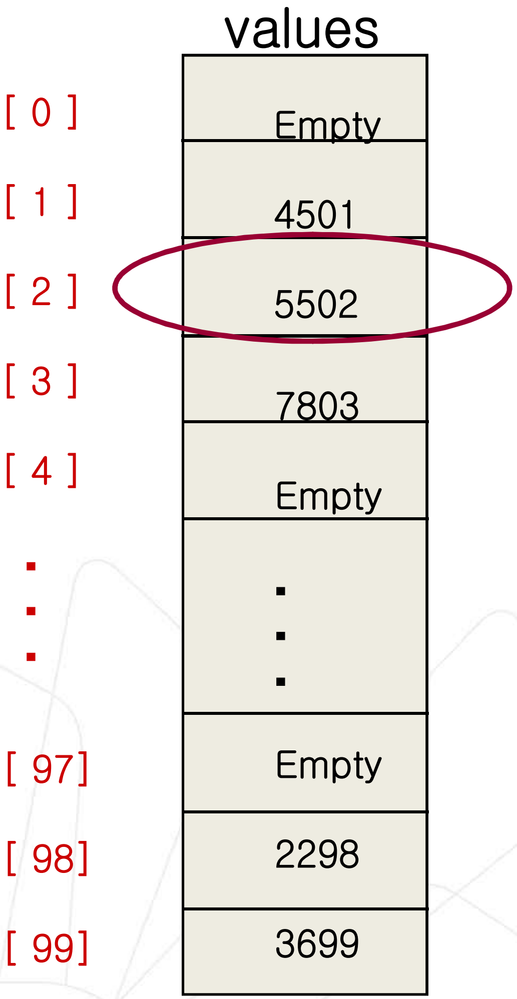
Hash(key) = partNum % 100

6702 % 100 = 2

But values[2] is already occupied.

COLLISION OCCURS

How to resolve the collision?



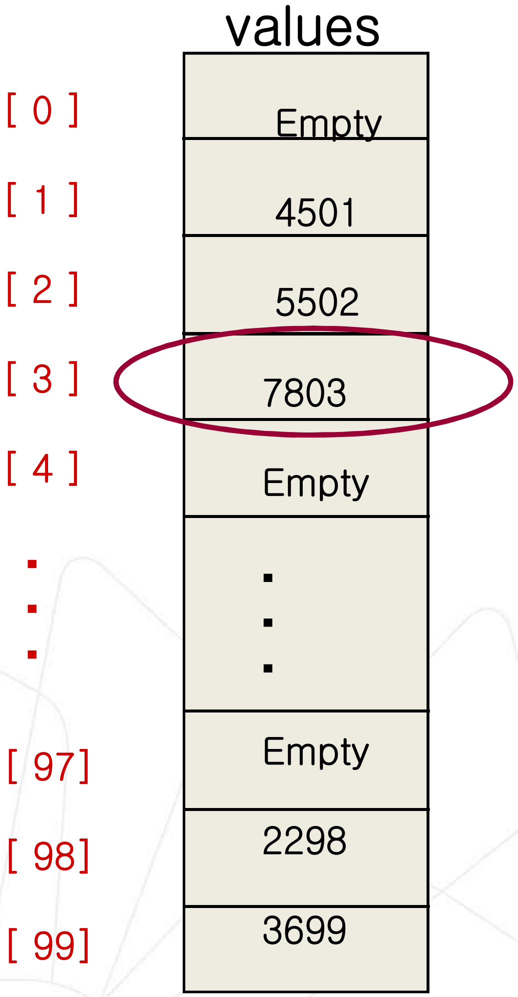
	values
[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

One way is by linear probing.
This uses the rehash function

(HashValue + 1) % 100

repeatedly until an empty location
is found for part number 6702.

Resolving the collision

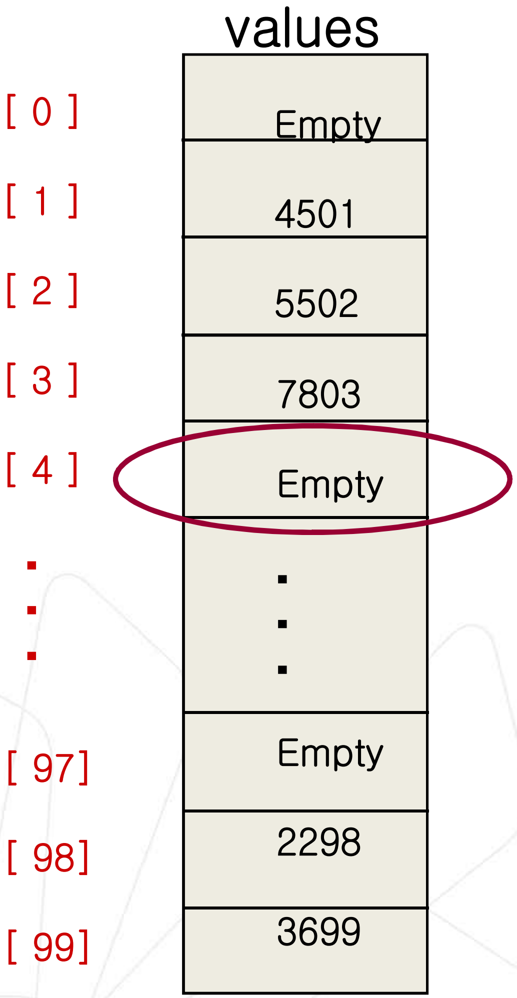


	values
[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

Still looking for a place for 6702 using the function

(HashValue + 1) % 100

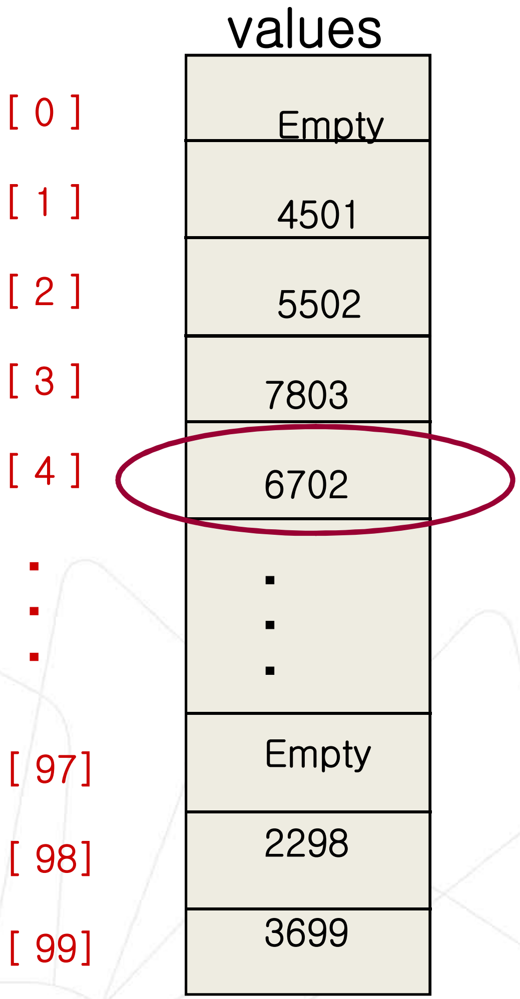
Collision resolved



	values
[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	Empty
⋮	⋮
[97]	Empty
[98]	2298
[99]	3699

Part 6702 can be placed at the location with index 4.

Collision resolved



	values
[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	6702
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

Part 6702 is placed at the location with index 4.

Where would the part with number 4598 be placed using linear probing?